

1: Introduction to the Web Standards Curriculum/Table of Contents

Introduction

For a while now, I've had a dream. My work in the last 8 or 9 years has been heavily focused around education, whether I've been commissioning and editing technical books to help people create cool stuff with technology, training new employees at the various companies I've worked for, or editing and writing tutorial articles to help people use Opera's software. I am passionate about the Web too, and a big believer in open web standards. I wanted to do my bit to help make the Web a better place, and I think this comes back to education, whether that's teaching people how to collaborate and have more respect for one another, or teaching them how to make their web sites work across platforms and devices, and be accessible to people with disabilities. Web standards are key to the latter, so I decided to try putting my time and energy into something that would help increase the adoption of web standards on the Web today and in the future. It has been floating around my head for a while now, but it has finally come to fruition at Opera—many thanks to my wonderful employers for paying me to do this! One of my dreams has finally been realised.

So in this article I introduce to you the product of a lot of hard work over the last several months (by myself and a lot of other people)—the Web Standards Curriculum, a course designed to give anyone a solid grounding in web design/development, no matter who they are—it is completely free to use, accessible, and assumes no previous knowledge. I am mainly aiming this at universities, as I believe the standards of education in web standards to be somewhat lacking at many universities. I've heard tales of students being marked down for using web standards in their coursework, because the marking schemes are so outdated; I've also heard tales of employers despairing because when they interview university graduates for web-related positions, they find out that the graduates really don't have a clue about real world web development. If you're at a progressive university that does teach web standards in a reasonable fashion, then I tip my hat to you—[get in touch!](#)

In this article I'll cover the following:

- [Why web standards?](#) Here I briefly discuss the advantages of using web standards, why they are not being adopted like they should, and how my course aims to tackle these issues
- [How the course is structured.](#) What it says on the tin; this section also talks about how educators should think about presenting the material to use it effectively in courses
- [Who should use this course?](#) When I say “anyone”, who do I mean, exactly?
- [The table of contents.](#) Skip to this bit if you're fed up with my waffle and want to get straight to the learning.
- [Acknowledgements](#)
- [Contact me](#)

Why web standards?

The main reasons that adopting web standards in your web design/development work is such a good idea are expanded on in article 4, but I'll go through them briefly here, to set the scene. Using web standards confers the following benefits:

1. **Efficiency of code:** As you'll learn throughout the course, a lot of best practice web standards usage is all about reusing code—you can separate your HTML content from your stylistic (CSS) and behavioural (JavaScript) information, allowing your file sizes to be kept small, and code to be written only once, and then reused wherever it is needed.
2. **Ease of maintenance:** This follows closely on from the last point—if you can write HTML only once, and then apply styles and behaviour wherever they are needed using classes and functions, then if you need to change something at a later date, you can just make the change in one place and it

propagates throughout the entire web site, rather than having to specify that change everywhere that it is needed!

3. **Accessibility:** The next two points are closely related—one of the big issues on the Web is *making web sites accessible to everyone, no matter who they are, regardless of circumstance*. This includes making web sites usable by people with disabilities such as blindness/visual impairment and motor impairment (ie, people who have restricted movement, and might not be able to use their hands properly, or at all). By using web standards and best practices, you'll be able to make your web sites usable by this significant group of the web audience with no extra effort.
4. **Device compatibility:** by this, I mean ensuring that your web sites will work not only across different platforms—ie Windows, Mac, Linux—but also alternative browsing devices, which these days can include mobile phones, TVs and games consoles. These devices have limitations such as screen size, processing power, control mechanisms available and more, but the good news is that again, using web standards and best practices, you can pretty much guarantee that your web sites will work on most of these devices. There are more mobile phones in the world than PCs, a lot of which are Internet-capable, so can you or your clients afford to miss out on this market? For more on mobile web development, check out some of the dedicated articles on dev.opera.com.
5. **Web crawlers/search engines:** By this, we are talking about what is termed *search engine optimization*—the practice of making your web sites as visible as possible to the so-called web crawlers that trawl the web and index web sites, and therefore giving you better search rankings on sites such as Google. There is a science to this (see SEO articles such as [Intelligent site structure for better SEO!](#) and [Semantic HTML and Search Engine Optimization](#)) but yet again, just by using web standards you will make your site a lot more visible on Google, Yahoo!, etc., which is good for business.

Even with all these advantages however, most sites on the Web still do not follow web standards, and many web developers working today still use bad, outdated practices. “Why?” You ask. There are a number of reasons for this—people will cite lack of education, company policy, not needing to learn standards because they are getting paid anyway, it’s too hard to learn, standards support in web browsers...let’s look at each one of these in more detail, and then look at the counter arguments, to try to get rid of any excuse for not adopting/learning standards.

1. **Lack of education:** There is an issue here, but this is one of the main reasons this course was created. A lot of universities don’t teach web standards in their web-related courses, and a lot of curriculums tend to contain outdated practices, and are hard to change due to bureaucracy. Books and training courses tend to be expensive. But wait! Now we’ve provided a course that’s free, and are running around universities etc to help make these changes for them, so there’s really now no excuse here.
2. **Company policy:** There is no doubt that some companies/institutions still have really old and outdated web sites. They may have policies that force their employees to use outdated browsers, but it is getting better, and now there is a free course available to easily show how to make changes, things should improve further. Upgrading a web site to modern standards encourages companies to upgrade the browsers that they use, as sites will not look as good in outdated browsers (although they should still work in older browsers). Companies should encourage their customers to upgrade as well. There is sound business reasoning as well—sites that use web standards, as explained above, will yield better search engine results and be accessible to people with disabilities and users of alternative devices—can companies afford to ignore this audience?
3. **“I don’t need to learn them!”:** I know some developers will sit there and say “but I’m using outdated practices and still getting paid—so why do I need to bother with this new stuff?” As explained above, it makes your code more efficient, easier to write, and easier to maintain. And it allows you to write modern code that is accessible and usable on alternative devices—isn’t that exciting? It will also make your skillset more future-proof, and make you capable of earning more. A lot of companies are requesting skills in web standards these days.
4. **“It’s too hard to learn!”:** Rubbish. After digesting some of this course, you’ll realize how easy it is to pick up the basics of using web standards, whether you’re new to web development/design, or an

existing web person upgrading your skillset. It is about as hard as using the old, outdated bad methods, which isn't very, and it confers so many advantages over the old ways.

5. Standards support in browsers: Standards support in browsers used to differ greatly, which made getting web sites to work across different browsers a nightmare. But those days are gone—modern browsers all have decent web standards support. Support is still sometimes needed for old browsers that don't have such good browser support, but by using modern best practices, you can ensure that users of those browsers will still have a reasonable user experience.

So as you can see, there's really not any excuse to not adopt web standards in your web development work. At least if you are coming to this course from the point of view of a beginner, you are starting off on the right foot and learning best practices from the start, rather than having to unlearn bad practices.

OK, so we keep talking about these bad practices in hushed tones, like they're the secret plans to the Death Star or something. We are not going to cover these practices in any detail in this course, as we don't believe we should; we think you should just be sent along the correct path to begin with. You are however probably wondering what they are, so let's just talk about them briefly.

In the old days, people used to do things like laying out their web sites inside giant tables, using the different table cells to position their graphics, text etc (not what tables were intended for, adds superfluous markup to the page). They used to use invisible images called spacer GIFs to fine tune positioning of page elements (not what images are intended for, add superfluous markup and images to the page). They used to write JavaScript that generated menus etc on the fly (no good for people with JavaScript disabled in their browsers, or people with visual impairments using screenreaders, which get confused by such JavaScript) or worked on only one browser (what about people using other browsers?). They used to insert styling information directly into the HTML using `` elements (terrible for maintenance, and adds superfluous markup to the page). And many other crimes against web development. The worst thing is that I say "in the old days" above, but the fact is that a lot of people are still doing things like this!

Web development is a messy skill at the best of times, but bad practices like these just make it harder. Using web standards and best practices, as outlined in this course, is the best way to go.

Course structure

The course is composed of several articles—there will be over 50 when the basic course is finalized—and each article is a few thousand words long. Each article focuses on a specific microtopic, and where appropriate, contains background on the topic, essential theory, practical examples and walkthrough tutorials to follow, and exercise questions to test your knowledge.

In addition to this we will make available a complete tutorial to follow in the future, which will go through the entire process of building a web site from start to finish.

A logical way to teach the course is to work out how many lessons you've got available to teach it over, and divide it by the number of articles. For each lesson, get the students to read over the articles connected to that lesson before the lesson occurs. Then go through practical examples during the lesson, and get the students to do the exercise questions after each lesson. Logically, I think an hour should be enough time to go through the concepts contained in each article, as long as you get the students to read each article before the lessons are taught. There is perhaps about 50 hours of teaching time in this course, and 50 hours of background reading.

Obviously, you'll have to think about the amount of time you teach the course over and exactly what to cover in each lesson, but experimentation is key.

Who should use this course?

This is a web standards course comprised of several articles, aimed at pretty much anyone who wants to learn web standards-based web design from scratch. It is intended to take the reader from nothing more than a basic familiarity with browsing the web, to being competent with CSS and HTML, and have basic

knowledge of JavaScript and how it fits in to the puzzle. It should give you enough knowledge to start thinking about entering the job market with confidence (obviously experience can't be taught).

Who is it aimed at? I want it to be usable by anyone who wants to learn web design "the right way":

1. **University/college students and teachers:** I have mentioned this already—this is an ideal set of articles to either create your own course from and deliver it to students, or use parts of to supplement your own course. To any students already studying some kind of web-related course, you should use this material to supplement your knowledge, and lobby your teachers into considering it as well! I would recommend all teachers/lecturers to look over this material as well, to make sure the techniques covered in their courses are current best practices.
2. **Pre-college/university age students:** While this course has been mainly written with adults in mind, there is no reason why younger students can't benefit from it—have a go and see how you get on.
3. **Existing web designers and developers:** There are a lot of existing web developers and designers out there who either aren't using web standards and best practices, or could use an easily accessible reference to look things up in, or use to brush up their knowledge. To the former, I urge you to give this course a chance and see how easy and valuable web standards are to adopt. To the latter, I'm sure you will find this course useful in helping others, brushing up on your skills, looking up hard-to-recall facts, and finding ammunition to help convince bosses and clients that things such as accessibility make a lot of sense.
4. **Educators inside companies:** This is an ideal way to provide inexpensive training to employees.
5. **Any other individuals:** If you are an individual who just fancies learning something about web design and development, then again, this is an inexpensive way to get some help with your endeavors. I am not expecting people to pay to use this course—it is released on a Creative Commons license, so freely available to anyone who wants to make use of it, as long as they give us the proper attribution.

Table of contents

Note that currently the first 39 articles of the curriculum are published, with roughly 10 more to follow to complete the course, asap.

The beginning

1. Introductory material, by Chris Mills—This is the one you're reading.

Introduction to the world of web standards

2. [The history of the Internet and the web, and the evolution of web standards](#), by Mark Norman Francis.
3. [How does the Internet work?](#), by Jonathan Lane.
4. [The Web standards model—HTML, CSS and JavaScript](#), by Jonathan Lane.
5. [Beautiful dream, but what's the reality?](#), by Jonathan Lane.

Web Design Concepts

This section won't go into any code or markup details, and will act as an introduction to the design process before you start to create any graphics or code, as well as concepts of web design such as IA, navigation, usability etc.

6. [Information Architecture—planning out a web site](#), by Jonathan Lane.
7. [What does a good web page need?](#), by Mark Norman Francis.
8. [Colour Theory](#), by Linda Goin.
9. [Building up a site wireframe](#), by Linda Goin.
10. [Colour schemes and design mockups](#), by Linda Goin.
11. [Typography on the web](#), by Paul Haine.

HTML basics

12. [The basics of HTML](#), by Mark Norman Francis.
13. [The HTML <head> element](#), by Christian Heilmann.

14. [Choosing the right doctype for your HTML documents](#), by Roger Johansson.

The HTML body

15. [Marking up textual content in HTML](#), by Mark Norman Francis.
16. [HTML Lists](#), by Ben Buchanan.
17. [Images in HTML](#), by Christian Heilmann.
18. [HTML links—let’s build a web!](#) by Christian Heilmann.
19. [HTML Tables](#), by Jen Hanen.
20. [HTML Forms—the basics](#), by Jen Hanen.
21. [Lesser-known semantic elements](#), by Mark Norman Francis.
22. [Generic containers—the div and span elements](#), by Mark Norman Francis.
23. [Creating multiple pages with navigation menus](#), by Christian Heilmann.
24. [Validating your HTML](#), by Mark Norman Francis.

Accessibility

25. [Accessibility basics](#), by Tom Hughes-Croucher.
26. [Accessibility testing](#), by Benjamin Hawkes-Lewis.

CSS

27. [CSS basics](#), by Christian Heilmann.
28. [Inheritance and Cascade](#), by Tommy Olsson.
29. [Text styling with CSS](#), by Ben Henick.
30. [The CSS layout model - boxes, borders, margins, padding](#), by Ben Henick.
31. [CSS background images](#), by Nicole Sullivan.
32. [Styling lists and links](#), by Ben Buchanan.
33. [Styling tables](#), by Ben Buchanan.
34. [Styling forms](#), by Ben Henick.
35. [Floats and clearing](#), by Tommy Olsson.
36. [CSS static and relative positioning](#), by Tommy Olsson.
37. [CSS absolute and fixed positioning](#), by Tommy Olsson.

JavaScript articles

To follow...

Supplementary articles

- [Getting your content online](#), by Craig Grannell.
- [More about the document <head>](#), by Chris Heilmann.
- [Supplementary: Common HTML entities used for typography](#), by Ben Henick.
- [The Opera Web Standards Curriculum glossary](#), by various authors. This is incomplete, and will be added to as time goes by.

Acknowledgements

The number of people who have helped me with this course are too numerous to mention in any great detail, but I have hopefully included everyone here. They are all great people, so check them out—go to their talks, buy their books, read their blogs, or do whatever else you can do to support them. I give to all of you my admiration and gratitude.

1. The authors: thank you so much to Ben Buchanan, Tom Hughes-Croucher, Mark Norman “Norm” Francis, Linda Goin, Paul Haine, Jen Hanen, Benjamin Hawkes-Lewis, Ben Henick, Christian Heilmann, Roger Johansson, Peter-Paul Koch, Jonathan Lane, Tommy Olsson, Nicole Sullivan, and Mike West. Without you, this course would be nothing, literally.
2. The Opera crew: best wishes to Jan Standal, David Storey, the rest of my team, and everyone else at Opera for believing in this idea, and helping me to develop the plan.
3. The organizations: thanks to everyone at Yahoo (the authors, and Sophie Major for helping to do a lot of organization and promotion), the WaSP (particularly Gareth Rushgrove, Stephanie Troeth and

Aarron Walter), the Britpack, the Geekup folks, and all the universities who showed an interest in looking at this course and helping to take it further.

4. The individuals: small mercies shall be granted to the following wonderful people—Craig Salla, Sara Dodd, John Allsopp, Roan Lavery, Bruce Lawson, Alan White. Sorry if I forgot anyone.
5. The readers: special hails to you for having an interest in creating web sites the right way, and taking time out to read this course!

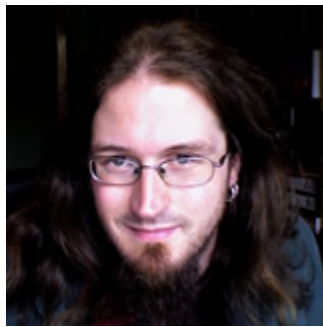
Contact me

I am constantly looking to improve this course, and get it adopted by as many people as possible. If you have any suggestions for how the course could be improved, any general comments to share, or want to talk to me about adopting it somewhere, then get in touch. My e-mail is [cmills \[at\] opera \[dot\] com](mailto:cmills@opera.com). You can also post comments about each article in the series using the “Discuss this article” link at the bottom of each one. You’ll need a [my.opera account](#) to participate in discussions.

[Next article—The history of the Internet and the web, and the evolution of web standards](#)

[Table of contents](#)

About the author



Chris Mills is a developer relations manager for Opera—he edits and publishes articles on [dev.opera.com](#) and [labs.opera.com](#), liaises with the community to raise awareness of Opera and collect feedback, and evangelises about Opera software wherever he can. He is also the organiser and editor of the Web Standards Curriculum.

Outside of work, he is an extremely avid music fan, enjoying playing and listening to a wide variety of music, including metal, folk, punk, electronica, prog, and more. His main band at the moment is the mighty [Conquest of Steel](#).

2: The history of the Internet and the web, and the evolution of web standards

BY [MNFRANCIS](#) · 8 JUL, 2008

Introduction

Where shall I begin, please your Majesty?

Begin at the beginning, the King said gravely, and go on till you come to the end: then stop.

Alice's Adventures in Wonderland; Lewis Carroll

Everything has to begin somewhere, so our journey will start with a focused history lesson. Below I am going to give you a brief overview of the creation of the Internet, the World Wide Web, and the web standards that this entire series focuses upon. I think it is useful and interesting to understand how we got to where we are, but it will be short enough so you don't get overwhelmed, and can get into the details nice and quickly. If any terms are unfamiliar to you, don't worry; if they're important for learning web development they'll be defined in the later articles that go into more depth on each subject, and you can always Google them! If you are already familiar with the history of the Internet or the World Wide Web, feel free to skip to the section on [web standards](#). The article contents are as follows:

- [The Internet's origins](#)
- [The creation of the world wide web](#)
 - [The browser wars](#)
- [The coming of web standards](#)
 - [The formation of the W3C](#)
 - [The web standards project](#)
 - [The rise of web standards](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

The Internet's origins

On the fourth of October in 1957 an event occurred that would change the world. The Soviet Union successfully launched the first satellite into Earth's orbit. Called Sputnik 1, it shocked the world—especially the United States of America, who had their own programme of satellite launches underway, but had yet to launch.

This event led directly to the creation of the US Department of Defence ARPA (the Advanced Research Projects Agency), due to a recognised need for an organisation that could research and develop advanced ideas and technology beyond the currently identified needs. Perhaps their most famous project (certainly the most widely used) was the creation of the Internet.

In 1960, psychologist and computer scientist Joseph Licklider published a paper entitled Man-Computer Symbiosis, which articulated the idea of networked computers providing advanced information storage and retrieval. In 1962, whilst working for ARPA as the head of the information processing office, he formed a group to further computer research, but left the group before any actual work was done on the idea.

The plan for this computer network (to be called ARPANET) was presented in October 1967, and in December 1969 the first four-computer network was up and running. The core problem in creating a network was how to connect separate physical networks without tying up network resources for constant links. The technique that solved this problem is known as packet switching and it involves data requests

being split into small chunks (packets,) which can be processed quickly without blocking communication from other parties—this principle is still used to run the Internet today.

This concept received wider adoption, with several other networks springing up using the same packet switching technique—for example, X.25 (developed by the International Telecommunication Union) formed the basis of the first UK university network JANET (allowing UK universities to send and receive files and emails) and the American public network CompuServe (a commercial enterprise allowing small companies and individuals access to time-shared computer resources, and then later Internet access.) These networks, despite having many connections, were more private networks than the Internet of today.

This proliferation of different networking protocols soon became a problem, when trying to get all the separate networks to communicate. There was a solution in sight however—Robert Kahn, whilst working on a satellite packet network project for ARPA, started defining some rules for a more open networking architecture to replace the current protocol used in ARPANET. Later joined by Vinton Cerf from Stanford University, the two created a system that masked the differences between networking protocols using a new standard. In the publication of the draft specification in December 1974, this was called the Internet Transmission Control Program.

This specification reduced the role of the network and moved the responsibility of maintaining transmission integrity to the host computer. The end result of this was that it became possible to easily join almost all networks together. ARPA funded development of the software, and in 1977 a successful demonstration of three different networks communicating was conducted. By 1981, the specification was finalised, published and adopted; and in 1982 the ARPANET connections outside of the US were converted to use the new TCP/IP protocol. The Internet as we know it had arrived.

The creation of World Wide Web

[Gopher](#) was an information retrieval system used in the early 1990s, providing a method of delivering menus of links to files, computer resources and other menus. These menus could cross the boundaries of the current computer and use the Internet to fetch menus from other systems. It was very popular with universities looking to provide campus-wide information and large organisations looking to centralise document storage and management.

Gopher was created by the University of Minnesota. In February, 1993, they announced that it was going to charge licensing fees for the use of their reference implementation of the Gopher server. As a consequence, many organisations started to look for alternatives to Gopher.

The European Council for Nuclear Research (CERN) in Switzerland had such an alternative. Tim Berners-Lee had been working on a information management system, in which text could contain links and references to other works, allowing the reader to quickly jump from document to document. He had created a server for publishing this style of document (called hypertext) as well as a program for reading them, which he had called WorldWideWeb. This software had first been released in 1991, however, it took two events to cause an explosion in popularity and the eventual replacement of Gopher.

On the thirtieth of April in 1993 CERN released the source code of WorldWideWeb into the public domain, so anyone could use or build upon the software without charge.

Then, later in the same year, the National Center for Supercomputing Applications (NCSA) released a program that was a combined web browser and Gopher client, called Mosaic. This was originally only available on Unix machines and in source code form, but in December 1993 Mosaic provided a new version with installers for both Apple Macintosh and Microsoft Windows. Mosaic rapidly increased in popularity, and with it the Web.

The number of available web browsers increased dramatically, many created by research projects at universities and corporations, such as Telenor (a Norwegian communications company,) which created the first version of the Opera browser in 1994.

The browser wars

The popularisation of the web brought commercial interests. Marc Andreessen left NCSA and together with Jim Clark founded Mosaic Communications, later renamed to Netscape Communications Corporation, and started work on what was to become Netscape Navigator. Version 1.0 of the software was released in December 1994.

Spyglass Inc. (the commercial arm of NCSA) licensed their Mosaic technology to Microsoft to form the basis of Internet Explorer. Version 1.0 was released in August 1995.

A rapid escalation soon followed, with Netscape and Microsoft each trying to get a competitive edge in terms of the features they support in order to attract developers. This has since become known as the browser wars. Opera maintained a small but steady presence throughout this period, and tried to innovate and support web standards as well as possible in these times.

The coming of web standards

During the browser wars, Microsoft and Netscape focused on implementing new features rather than on fixing problems with the features they already supported, and adding proprietary features and creating features that were in direct competition with existing features in the other browser, but implemented in an incompatible way.

Developers at the time were forced to deal with ever increasing levels of confusion when trying to build web sites, sometimes to the extent of building two different but effectively duplicate sites for the two main browsers, and other times just choosing to support only one browser, and blocking others from using their sites. This was a terrible way of working, and the inevitable backlash from developers was not far away.

The formation of the W3C

In 1994, Tim Berners-Lee founded the World Wide Web Consortium (W3C) at the Massachusetts Institute of Technology, with support from CERN, DARPA (as ARPA had been renamed to) and the European Commission. The W3C's vision was to standardize the protocols and technologies used to build the web such that the content would be available to as wide a population of the world as possible.

During the next few years, the W3C published several specifications (called recommendations) including HTML 4.0, the format for PNG images, and Cascading Style Sheets versions 1 and 2.

However, the W3C do not enforce their recommendations. Manufacturers only have to conform to the W3C documents if they wish to label their product as W3C-compliant. In practice, this is not a valuable selling point as almost all users of the web do not know, nor probably care, who the W3C are. Consequently, the browser wars continued unabated.

The Web Standards Project

In 1998, the browser market was dominated by Internet Explorer 4 and Netscape Navigator 4. A beta version of Internet Explorer 5 was released, and it implemented a new and proprietary dynamic HTML. This meant that professional web developers needed to know five *different* ways of writing JavaScript.

As a result, a group of professional web developers and designers banded together. This group called themselves the Web Standards Project (WaSP). The idea was that by calling the W3C documents standards rather than recommendations, they might be able to convince Microsoft and Netscape to support them.

The early method of spreading the call to action was done using a traditional advertising technique called a roadblock, where a company would take out an advert on all broadcast channels at the same time, so no matter how a viewer would flick between channels, they would get exactly the same message. The WaSP published an article simultaneously on various web development focused sites including builder.com, Wired online, and some popular mailing lists.

Another technique they used was to ridicule the companies that would join the W3C (and other standards bodies) but then focus more on creating new features than on getting the basics *that they had signed up for* correct to start with.

This all sounds a bit negative, but the WaSP didn't just sit there criticising people—they also helped. Seven members formed the CSS Samurai, who identified the top ten problems with the CSS support in Opera and Internet Explorer (Opera fixed their problems, Microsoft did not).

The rise of web standards

In 2000, Microsoft released Internet Explorer 5 Macintosh Edition. This was a very important milestone, it being the default browser installed with the Mac OS at the time, and having a reasonable level of support for the W3C recommendations too. Along with Opera's decent level of support for CSS and HTML, it contributed to a general positive movement, where web developers and designers felt comfortable designing sites using web standards for the first time.

The WaSP persuaded Netscape to postpone the release of the 5.0 version of Netscape Navigator until it was much more compliant (this work formed the basis of what is now Firefox, a very popular browser). The WaSP also created a Dreamweaver Task Force to encourage Macromedia to change their popular web authoring tool to encourage and support the creation of compliant sites.

The popular web development site A List Apart was redesigned early in 2001 and in an article describing how and why, stated:

In six months, a year, or two years at most, all sites will be designed with these standards. [...] We can watch our skills grow obsolete, or start learning standards-based techniques now.

That was a little optimistic—not all sites, even in 2008, are built with web standards. But many people listened. Older browsers decreased in market share, and two more very high profile sites redesigned using web standards: Wired magazine in 2002, and ESPN in 2003 became field leaders in supporting web standards and new techniques.

Also in 2003, Dave Shea launched a site called the CSS Zen Garden. This was to have more impact on web professionals than anything else, by truly illustrating that the entire design can change just by changing the style of the page; the content could remain identical.

Since then in the professional web development community web standards have become *de rigeur*. And in this series, we will give you an excellent grounding in these techniques so that you can develop websites just as clean, semantic, accessible and standards-compliant as the big companies'.

Summary

In this article I've looked at how the modern Internet was created as a result of the space race; how Tim Berners-Lee defined hypertext for a generation and how the commercial interests of two companies caused one of the most notable developer backlashes ever seen. The term web standards is now more widely used by web professionals than any other term applied by the W3C (in fact the W3C have started to use the term on their own pages), so that is what we are going to teach you—the *standards* way to build web sites.

Further reading

If you want to know more, you may like to visit some of the following sites:

- [The history of the Internet \(wikipedia\)](#)
- [The history of the World Wide Web \(wikipedia\)](#)
- [The history of the W3C](#)
- [The Web Standards Project](#), and their [history](#)

- [A List Apart](#)
- [CSS Zen Garden](#)

Exercise questions

Or you might like to try researching further, by answering these questions:

- What browsers are available on the Internet today for users of Windows, Mac OS X and Linux?
- What percentage of web users use each browser?
- What browsers do mobile devices use when accessing web pages?
- How many web standards have the W3C published, and which are widely supported by browser manufacturers today?

About the author



Photo credit: [Andy Budd](#).

Mark Norman Francis has been working with the internet since before the web was invented. He currently works at Yahoo! as a Front End Architect for the world's biggest website, defining best practices, coding standards and quality in web development internationally.

Previous to Yahoo! he worked at Formula One Management, Purple Interactive and City University in various roles including web development, backend CGI programming and systems architecture. He pretends to blog at <http://marknormanfrancis.com/>.

3: How does the Internet work?

BY [JONATHAN LANE](#) · 8 JUL, 2008

Introduction

Every so often, you get offered a behind-the-scenes look at the cogs and fan belts behind the action. Today's your lucky day. I'm going to usher you behind the scenes of one of the hottest technologies that you might already be familiar with: the World Wide Web. Cue theme music.

This article covers the underlying technologies that power the World Wide Web:

- Hypertext Markup Language (HTML)
- Hypertext Transfer Protocol (HTTP)
- Domain Name System (DNS)
- Web servers and web browsers
- Static and dynamic content

It's all pretty fundamental stuff—while most of what's covered here won't help you to build a better web site, it will give you the proper language to use when speaking with clients and with others about the web. It's like a wise nun-turned-nanny once said in *The Sound of Music*: “When we read we begin with ABC. When we sing we begin with Do Re Mi.”. In this article I will briefly look at how computers actually communicate using HTTP and TCP/IP, then go on to look at the different languages that go together to create the web pages that make up the Internet. The contents of this article are as follows:

- [How do computers communicate via the Internet?](#)
 - [Dissecting a request/response cycle](#)
- [Types of content](#)
 - [Plain text](#)
 - [Web standards](#)
 - [Dynamic web pages](#)
 - [Formats requiring other applications or plugins](#)
- [Static vs. dynamic web sites](#)
- [Summary](#)
- [Exercise questions](#)

How do computers communicate via the Internet?

Thankfully, we have kept things simple for computers. When it comes to the World Wide Web, most pages are written using the same language, HTML, which is passed around using a common protocol—HTTP (hypertext transfer protocol). HTTP is the common internet dialect (specification), allowing for example a Windows machine to sing in harmony with a machine running the latest and greatest version of Linux (Do Re Mi!). Through the use of a web browser—a special piece of software that interprets HTTP and renders HTML into a human-readable form—web pages authored in HTML on any type of computer can be read anywhere, including telephones, PDAs and even popular video game systems.

Even though they're speaking the same language, the various devices accessing the web need to have some rules in place to be able to talk to one another—it's like learning to raise your hand to ask a question in class. HTTP lays out these ground rules for the Internet. Because of HTTP, a client machine (like your computer) knows that it has to be the one to initiate a request for a web page from a server. A server is a computer where web sites reside - when you type a web address into your browser, a server receives your request, finds the web page you want, and sends it back to your computer to be displayed in your web browser.

Dissecting a request/response cycle

Now that I've looked at all the parts that allow computers to communicate across the Internet, I'll look at the HTTP request/response cycle in more detail. There are some numbered steps below for you to work along with, so I can demonstrate some of the concepts to you more effectively.

1. Every request/response starts by typing a Universal Resource Locator (URL) into the address bar of your web browser, something like `http://dev.opera.com`. Open a browser and do this now.

Now, one thing you may not know is that web browsers actually don't use URLs to request web sites from servers; they use Internet Protocol or IP addresses (which are basically like phone numbers or postal addresses that identify servers.) For example, the IP address of `http://dev.opera.com` is 213.236.208.98.

2. Try opening a new browser tab or window, typing `http://www.apple.com` and hitting enter; then type `http://17.149.160.10/` and hit enter—you will get to the same place. Try typing `http://213.236.208.98` into the address bar and hitting enter—you will get to the same server location that you got to in step 1, although you'll get a 403 "Access Denied" error—this is because you don't have permission to access the actual root of this server.

`http://www.apple.com` is basically acting as an alias for `http://17.149.160.10/`, but why, and how? This is because people are better at remembering words than long strings of numbers. The system that makes this work is called the Domain name system (DNS), which is essentially a comprehensive automatic directory of all of the machines connected to the Internet. When you punch `http://dev.opera.com` into your address bar and hit enter, that address is sent off to a name server that tries to associate it to its IP address. There are a ton of machines connected to the Internet, and not every DNS server has a listing for every machine online, so there's a system in place where your request can get referred on to the right server to fulfill your request.

So the DNS system looks up the `www.opera.com` web site, finds that it is located at 17.149.160.10, and sends this IP address back to your web browser.

Your machine sends a request to the machine at the IP address specified and waits to get a response back. If all goes well, the server machine sends a short message back to the client with a message saying that everything is okay (see Figure 1,) followed by the web page itself. This type of message is contained in an HTTP header.

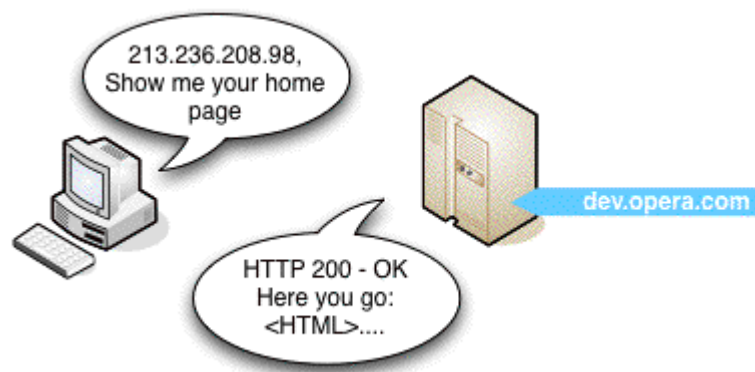


Figure 1: In this case, everything is fine, and the server returns the correct web page.

If something goes wrong, for example you typed the URL incorrectly, you'll get an HTTP error returned to your web browser instead—the infamous 404 "page not found" error is the most common example you'll come across.

3. Try typing in `http://dev.opera.com/joniscool.html`—the page doesn't exist, so you'll get a 404 error returned. Try it with a few pages, on different web sites, that don't exist, and you'll see a variety of different pages returned. This is because some web developers have just left the web server

to return their default error pages, and others have coded custom error pages to appear when a non-existent page is returned. This is an advanced technique that won't be covered in this course, but it will hopefully be covered in a separate dev.opera.com article soon.

Lastly, a note about URLs—usually the first URL you go to on a site doesn't have an actual file name at the end of it (eg `http://www.mysite.com/`), and then subsequent pages sometimes do and sometimes don't. You are always accessing actual files, but sometimes the web developer has set up the web server to not display the file names in the URL—this often makes for neater, easier to remember URLs, which leads to a better experience for the user of your web site. We'll not cover how to do this in this course, as again, it is quite advanced; [we cover uploading files to a server and file/folder directory structures in a later article](#).

Types of Content

Now that I've shown you an HTTP request/response, I'll now turn your attention to the different types of content you'll expect to see on the Internet. I've grouped these into 4 types—plain text, web standards, dynamic web pages, and formats requiring other applications or plugins.

Plain text

In the really early days of the Internet, before any web standards or plugins came along, the Internet was mainly just images and plain text—files with an extension of .txt or similar. When a plain text file is encountered on the Internet, the browser will just display it as is, without any processing involved. You often still get plain text files on university sites.

Web Standards

The basic building blocks of the World Wide Web are the three main web standards—HTML (or XHTML, I'll use the two interchangeably here for our purposes), CSS and JavaScript.

Hypertext Markup Language is actually a pretty good name as far as communicating it's purpose. HTML is what's used to divide up a document, specify its contents and structure, and define the meaning of each part (it's what contains all the text etc that you see on web sites.) It uses elements to identify the different components of a page.

Cascading Style Sheets give you complete control over how an element is displayed. It's easy, using style declarations, to change all paragraphs to be double-spaced (`line-height: 2em;`), or to make all second-level headings green (`color: green;`). There are a ton of advantages to separating the structure from the formatting, and we'll look at this in more detail [in the next article](#). To demonstrate the power of HTML and CSS used together, Figure 2 shows some plain HTML on the left, with no formatting added to it at all, while on the right you can exactly the same HTML with some CSS styles applied to it.

Example page to show CSS styling

Web browsers will apply some basic formatting to an HTML document without any style declarations.

CSS ability

Cascading Style Sheets allow you to control the appearance of any element within your HTML document. You can, for example, change font sizes, colors, backgrounds, add borders or spacing in and around elements.

EXAMPLE PAGE TO SHOW CSS STYLING

Web browsers will apply some basic formatting to an HTML document without any style declarations.

CSS ability

Cascading Style Sheets allow you to control the appearance of any element within your HTML document. You can, for example, change font sizes, colors, backgrounds, add borders or spacing in and around elements.

Figure 2: Plain HTML on the left, HTML with CSS applied to it on the right.

Finally, JavaScript provides dynamic functions to your web site. You can write small programs in JavaScript that will run on the client computer, requiring no special software to be installed on the server. JavaScript allows you to add some basic functionality and interactivity to your web site, but it has its limitations, which brings us to server-side programming languages, and dynamic web pages.

Dynamic web pages

Sometimes, when browsing the Internet, you'll come across web pages that don't have an .html extension—they might have a .php, .asp, .aspx, .jsp, or some other strange extension. These are all examples of dynamic web technologies, which can be used to create web pages that have dynamic sections—code that displays different results depending on values fed to it, eg from a database, form, or other data source. We'll cover these types of web pages in the Static versus Dynamic pages section below.

Formats requiring other applications or plugins

Because web browsers are only equipped to interpret and display certain technologies like web standards, if you've requested a URL that points to either a complex file format, or a web page containing a technology requiring plugins, it will either be downloaded to your computer or opened using the required plugin if the browser has it installed. For example:

1. If you encounter a Word document, Excel file, PDF, compressed file (ZIP, or SIT for example,) complex image file such as a Photoshop PSD, or another complex file that the browser doesn't understand, the browser will usually ask you if you want to download or open the file. Both of these usually have similar results, except that the latter will cause the file to be downloaded and then opened by an application that does understand it, if one is installed.
2. If you encounter a page containing a Flash movie, MP3 or other music format, MPEG or other video format, the browser will play it using an installed plugin, if one has been installed. If not, you will either be given a link to install the required plugin, or the file will download and look for a desktop application to run it.

Of course, there are some gray areas—for example SVG (Scalable Vector Graphics) is a web standard that runs natively in some browsers, such as Opera, but not in others, such as Internet Explorer—IE needs a plugin to understand SVG. A number of browsers will come with some plugins pre-installed, so you may not be aware that content is being displayed via a plugin and not natively within the browser.

Static vs. Dynamic Web Sites

So what are static and dynamic web sites, and what is the difference between the two? Similar to a box of chocolates, it's all in the filling:

A static web site is a web site where the content, the HTML and graphics, are always static—it is served up to any visitor the same, unless the person who created the web site decides to manually change the copy of it on the server—this is exactly what we've been looking at throughout most of this article.

On a dynamic web site on the other hand, the content on the server is the same, but instead of just being HTML, it also contains dynamic code, which may display different data depending on information you feed to the web site. Let's look at an example—navigate to www.amazon.com in your web browser, and search for 5 different products. Amazon hasn't sent you 5 different pages; it has sent you the same page 5 times, but with different dynamic information filled in each time. This different information is kept in a database, which pulls up the relevant information when requested, and gives it to the web server to insert in the dynamic page.

Another thing to note is that special software must be installed on the server to create a dynamic web site. Whereas normal static HTML files are saved with a file extension of .html, these files contain special dynamic code in addition to HTML, and are saved with special file extensions to tell the web server that they need extra processing before they are sent to the client (such as having the data inserted from the database)—PHP files for example usually have a .php file extension.

There are many dynamic languages to choose from—I've already mentioned PHP, and other examples include Python, Ruby on Rails, ASP.NET and Coldfusion. In the end, all of these languages have pretty much the same capabilities, like talking to databases, validating information entered into forms, etc., but they do things slightly differently, and have some advantages and disadvantages. It all boils down to what suits you best.

We won't be covering dynamic languages any further in this course, but I have provided a list of resources here in case you want to go and read up on them:

- Rails: Fernandez, Obie. (2007), The Rails Way. Addison-Wesley Professional Ruby Series.
- [Rails screencasts](#)
- PHP: Powers, David (2006), PHP Solutions: Dynamic web development made easy, friends of ED.
- [PHP Online documentation](#)
- ASP.NET: Lorenz, Patrick. (2003). ASP.NET 2.0 Revealed. Apress.
- ASP.NET: [online ASP.NET documentation and tutorials](#).

Summary

That's it for the behind-the-scenes tour of how the Internet works. This article really just scratches the surface of a lot of the topics covered, but it is useful as it puts them all in perspective to each other, showing how they all relate and work together. There is still a lot left to learn about the actual language syntax that makes up HTML, CSS and JavaScript, and this is where we'll go to next—the next article focuses on the HTML, CSS and JavaScript “web standards” model of web development, and takes a look at web page code.

Exercise questions

- Provide a brief definition for HTML and HTTP and explain the difference between the two.
- Explain the function of a web browser.
- Have a look around the Internet for about 5-10 minutes and try to find some different types of content—plain text, images, HTML, dynamic pages such as PHP and .NET (.aspx) pages, PDFs, word documents, Flash movies etc. Access some of these and have a think about how your computer displays them to you.
- What is the difference between a static page and a dynamic page?
- Find a list of HTTP error codes, list 5 of them, and explain what each one means.

About the author



Jonathan Lane is the President of [Industry Interactive](#)—a web development/web application development company located on Mayne Island, British Columbia, Canada. He got his start in development working for the University of Lethbridge Curriculum Re-Development Center as their web projects coordinator for many years.

He blogs at [Flyingtroll](#) and is currently developing [Mailmanagr](#), an e-mail interface for the Basecamp project management application.

4: The Web standards model - HTML, CSS and JavaScript

BY [JONATHAN LANE](#) · 8 JUL, 2008

Introduction

In the last article, we touched briefly on the basic building blocks of the web—HTML (or XHTML), CSS and JavaScript. Now it's time for me to dig a little deeper and to look at each of these—what they do, and how the three interact with each other to create a web site. The contents of this article are as follows:

- [Why separate?](#)
- [Markup, the basis of every page](#)
 - [Is XHTML the adult-rated version?](#)
 - [Validation, what's that?](#)
- [CSS—let's add some style](#)
- [JavaScript—adding behaviour to web pages](#)
- [An example page<](#)
 - [index.html](#)
 - [styles.css](#)
- [Summary](#)
- [Exercise questions](#)

Why separate?

That's usually the first question that gets asked about web standards. You can accomplish content, styling and layout just using HTML—font elements for style and HTML tables for layout, so why should I bother with this XHTML/CSS stuff? Tables for layout, etc. is how it used to be done in the bad old days of web design, and many people still do it like this (although you really shouldn't be), which is one of the reasons why we created this course in the first place. We won't be covering such methods in this course. Here are the most compelling reasons for using CSS and HTML over outdated methods:

1. **Efficiency of code:** The larger your files are, the longer they will take to download, and the more they will cost some people (some people still pay for downloads by the megabyte.) You therefore don't want to waste your bandwidth on large pages cluttered up with styling and layout information in every HTML file. A much better alternative is to make the HTML files stripped down and neat, and include the styling and layout information just once in separate CSS file(s.) To see an actual case of this in action, check out [the A List Apart Slashdot rewrite article](#) where the author took a very popular web site and re-wrote it in XHTML/CSS.
2. **Ease of maintenance:** Following on from the last point, if your styling and layout information is only specified in one place, it means you only have to make updates in one place if you want to change your site's appearance. Would you prefer to update this information on every page of your site? I didn't think so.
3. **Accessibility:** Web users who are visually impaired can use a piece of software known as a "screen reader" to access the information through sound rather than sight—it literally reads the page out to them. In addition voice input software, used by people with mobility impairments, also benefits from well constructed semantic web pages. Much like a screen reader user uses keyboard commands to navigate headings, links and forms, a voice input user will use voice commands. Web documents marked up semantically rather than presentationally can be easier to navigate and the information in them is more likely to be found by the user. In other words, the faster you "get to the point" (the content), the better. Screen readers can't access text locked away in images, and find some uses of JavaScript confusing. Make sure that your critical content is available to everyone.
4. **Device compatibility:** Because your XHTML page is just plain markup, with no style information, it can be reformatted for different devices with vastly differing attributes (eg screen size) by simply applying an alternative style sheet—you can do this in a few different ways (look at the [mobile articles on dev.opera.com](#) for resources on this). CSS also natively allows you to specify different style sheets

for different presentation methods/media types (eg viewing on the screen, printing out, viewing on a mobile device.)

5. Web crawlers/search engines: Chances are you will want your pages to be easy to find by searching on Google, or other search engines. A search engine uses a “crawler”, which is a specialized piece of software, to read through your pages. If that crawler has trouble finding the content of your pages, or mis-interprets what’s important because you haven’t defined headings as headings and so on, then chances are your rank in the search results will suffer.
6. It’s just good practice: This is a bit of a “because I said so” reason, but talk to any professional standards-aware web developer or designer, and they’ll tell you that separating content, style, and behaviour is the best way to develop an application.

Markup, the basis of every page

HTML and XHTML are markup languages composed of elements, which contain attributes (some optional and some mandatory.) These elements are used to markup the various different types of content in documents, specifying what each bit of content is supposed to be rendered as in web browsers (for example headings, paragraphs, tables, bulleted lists etc.

As you’d expect, elements define the actual content type, while attributes define extra information about those elements, such as an ID to identify that element, or a location for a link to point to. You should bear in mind that markup is supposed to be as semantic as possible, ie it is supposed to describe the function of the content as accurately as possible. Figure 1 shows the anatomy of a typical (X)HTML element.

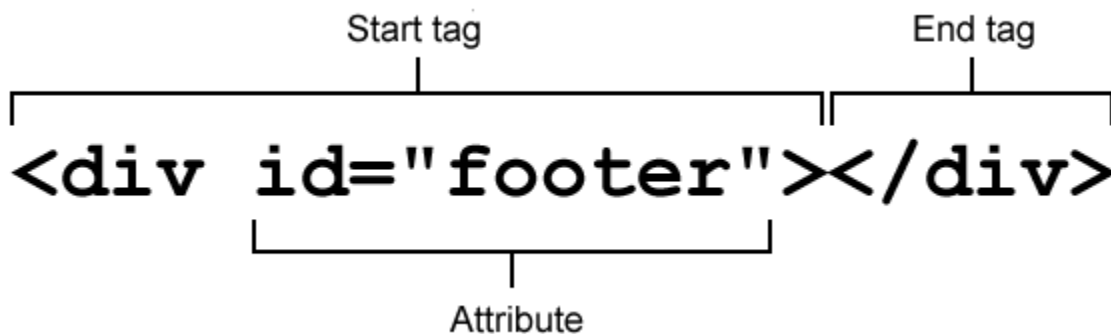


Figure 1: Anatomy of an (X)HTML element. [Read Figure 1 description](#)

With that in mind, just what is the difference between HTML and XHTML?

What is XHTML?

The “X” in XHTML means “extensible”. One of the most common questions for those starting out is “should I be using HTML or XHTML, and what the heck is the difference?”. They pretty much do the same thing; the biggest difference is in the structure. See Table 1 for the main differences.

HTML	XHTML
Elements and attributes are case insensitive, <code><h1></code> is the same thing as <code><H1></code> .	Elements and attributes are case sensitive; they are all lowercase.
Certain elements don’t need a closing tag (eg paragraphs, <code><p></code>), while others (called “empty elements”) forbid the closing tag	All elements must be explicitly closed (eg <code><p>A paragraph</p></code>). Elements without content may be closed using a slash in the start tag (eg <code><hr></hr></code> and <code><hr /></code> mean the same thing). If you are serving your XHTML as <code>text/html</code> , then you should use the shorthand syntax on all elements that are defined as being “empty” and place a space before the slash. You should

(eg images,).	use the long form (with separate start and end tags) on any element not defined as empty—even if you don't have any content in it.
Some attribute values may be written without being enclosed in quotes.	Attribute values must be enclosed by quotes.
Shorthand can be used for certain attributes (ie <option selected>).	The full attribute form must be used for all attributes (eg <option selected="selected">).
Servers should deliver HTML to the client with a media type of text/html.	XHTML Should use the application/xhtml+xml media type but may use application/xml, text/xml or text/html. If text/html is used then the HTML compatibility guidelines should be followed because browsers will treat it as HTML (and use error recovery to account for the differences between the languages).

Table 1: Differences between HTML and XHTML.

For now, we'd recommend that you don't worry too much about whether you are writing HTML or XHTML. Stick to the advice presented throughout this course and use an HTML doctype ([see article 14 for more on doctypes](#)) and you shouldn't go far wrong.

Validation, what's that?

Because HTML and XHTML are set standards (and CSS too, for that matter), the World Wide Web Consortium (W3C) has created a great tool called a validator to automatically check your pages for you, and point out any problems/errors your code might have, such as missing closing tags or missing quotes around attributes. The HTML validator is available online at <http://validator.w3.org/>. It will automatically detect whether you're using HTML or XHTML, and which doctype you're using. If you want to check out your CSS, the validator for that is available at <http://jigsaw.w3.org/css-validator/>.

For more information on validation, see Article 24 (not yet published). For more information on doctypes, see article [Article 14](#).

CSS—let's add some style

Cascading Style Sheets allow you fine control over the formatting and layout of your document. You can change or add colors, backgrounds, font sizes and styles, and even position things on your web page in different places. There are 3 main ways to apply styles using CSS: redefining an element, applying a style to an ID, or applying a style to a class. Let's take a look at each:

1. Redefining an element. You can change the way any (X)HTML element displays by defining a rule to style it. If you want all of your paragraphs to be double-spaced and green, in CSS you can add in this declaration:

```
2.    p {
3.        line-height: 2;
4.        color: green;

    }
```

Now any content enclosed within <p></p> tags will have double the line height, and be colored green.

5. Defining an ID. You can give an element an id attribute to uniquely identify it on a page (each ID can be used only once on a page)—for example id="navigation_menu". This lets you have finer control over formatting on a page, for example, if you only want a certain paragraph double-spaced and highlighted with green text, give it an ID:

```
<p id="highlight">Paragraph content</p>
```

And then apply a CSS rule to it like follows:

```
#highlight {  
  line-height: 2;  
  color: green;  
}
```

This will only apply the CSS rule to the paragraph on the page with an `id` attribute of `highlight` (the pound sign is just CSS convention to indicate that it's an ID).

6. Defining a class. Classes are just like IDs, except that you can have more than one of the same class on each page. Following along with our double-spacing example, if you want to double space and highlight the first two paragraphs on a page, you would add classes to them like so:

7.

```
<p class="highlight">Paragraph content</p>
```

```
<p class="highlight">The content of the second paragraph</p>
```

And then apply a CSS rule to them like follows:

```
.highlight {  
  line-height: 2;  
  color: green;  
}
```

`highlight` is a class this time, not an ID—the period is just CSS convention to indicate that it's a class.

The example below will give you more of an idea of how CSS styles HTML; we'll start looking at CSS in way more detail in Article 22, which will be published soon.

JavaScript—adding behaviour to web pages

Finally, JavaScript is the scripting language that you use to add behaviour to your web pages—it can be used to validate the data you enter into a form (tell you if it is in the right format or not), provide drag and drop functionality, change styles on the fly, animate page elements such as menus, handle button functionality, and a million other things. Most modern JavaScript works by finding a target HTML element, and then doing something to it, just like CSS, but the way it operates, the syntax etc is rather different.

JavaScript is a more complicated and expansive subject than HTML and CSS, so to keep things simple and avoid confusion at this stage, I won't be discussing it in the below example. In fact, you won't be looking at JavaScript in this course again until much later on.

An example page

There are a lot of details I haven't covered here, but we'll get through everything during this web design course! For now, I'll present you with a real page example, just to give you a taste of what you'll be working with in the rest of the articles.

The example I present below is a references page, which you could use to cite references at the end of say, a psychology essay on the group dynamics of a web development team, or a report for work on broadband Internet use in the United States. Please note, that if you're a stickler for strict academic writing, this example shows APA formatting (I had to pick one). [Download the code here](#).

[index.html](#)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

  <title>References</title>
  <style type="text/css">
    @import url("styles.css");
  </style>
</head>

<body>
  <div id="bggraphic"></div>
  <div id="header">
    <h1>References</h1>
  </div>
  <div id="references">
    <cite class="article">Adams, J. R. (2008). The Benefits of Valid Markup: A
    Post-Modernistic Approach to Developing Web Sites. <em>The Journal of Awesome Web
    Standards, 15:7,</em> 57-62.</cite>
    <cite class="book">Baker, S. (2006). <em>Validate Your Pages.... Or
    Else!</em> Detroit, MI: Are you out of your mind publishers.</cite>
    <cite class="article">Lane, J. C. (2007). Dude, HTML 4, that's like so 2000.
    <em>The Journal that Publishes Genius, 1:2, </em> 12-34.</cite>
    <cite class="website">Smith, J. Q. (2005). <em>Web Standards and You.</em>
    Retrieved May 3, 2007 from Web standards and you.</cite>
  </div>
  <div id="footer">
    <p>The content of this page is copyright © 2007 <a
    href="mailto:jonathanlane@gmail.com">J. Lane</a></p>
  </div>
</body>
</html>
```

I'm not going to dissect this file line by line, as you'll see many examples in future articles, however, a few major things to take note of are as follows.

Line 1 is what's called the document type declaration, or doctype. In this case, the page is XHTML 1.0 Transitional. The doctype specifies a set of rules that your markup has to follow, and can be validated against. See [Article 14](#) for more on doctypes.

Lines 5 to 7 import a CSS file into the page—the styles contained in this file will be applied to the various elements on the page. You'll see the content of the CSS file that handles all of the formatting for the page in the next section.

I've assigned a different class to each of the different types of reference. Doing this lets me apply a different style to each type of reference—for instance in our example I've put a different color to the right of each reference to make it easier to scan the list.

Now let's take a look at the CSS that styles the HTML.

styles.css

```
body {
  background: #fff url('images/gradbg.jpg') top left repeat-x;
  color: #000;
  margin: 0;
  padding: 0;
  border: 0;
  font-family: Verdana, Arial, sans-serif; font-size: 1em;
```

```
}

div {
  width: 800px;
  margin: 0 auto;
}

#bggraphic {
  background: url('images/pen.png') top left no-repeat;
  height: 278px;
  width: 362px;
  position: absolute;
  left: 50%;
  z-index: -100;
}

h1 {
  text-align: center;
  text-transform: uppercase;
  font-size: 1.5em;
  margin-bottom: 30px;
  background: url('images/headbg.png') top left repeat;
  padding: 10px 0;
}

#references cite {
  margin: 1em 0 0 3em;
  text-indent: -3em;
  display: block;
  font-style: normal;
  padding-right: 3px;
}

.website {
  border-right: 5px solid blue;
}

.book {
  border-right: 5px solid red;
}

.article {
  border-right: 5px solid green;
}

#footer {
  font-size: 0.5em;
  border-top: 1px solid #000;
  margin-top: 20px;
}

#footer a {
  color: #000;
  text-decoration: none;
}

#footer a:hover{
  text-decoration: underline;
}
```

I went a little overboard with styling up this page, adding some neat background effects in order to show you some of the things that can be accomplished using CSS.

Line 1 sets some defaults for the document such as text and background color, width of border to add around the text, etc. Some people won't bother to explicitly state defaults like these, and most modern browsers will apply their own defaults, but it is a good idea, as it allows you more control over how your web site will look across different browsers.

On the next line I've set the page to be 800px wide (although I could have specified a percentage here to have the page expand/contract based on the size of the browser window. The margin setting I've used here will ensure that the page content stays centered in the window.

Next let's turn our attention to the background images used in the page (these are applied using the background: url declarations.) I've got 3 different background elements on this page. The first is a gradient that tiles across the top of the page giving us the nice blue fade. Second, I've used a semi-transparent PNG for the pen graphic in order to provide enough contrast with the text above it and to pick up the gradient (semi-transparent PNG images don't work in Internet Explorer 6 or below, but they work in pretty much every modern browser; see [Dean Edward's IE-fixing JavaScript](#) for an IE6 solution to this issue, which also fixes some of IE6's CSS support issues.) Finally, I've used another semi-transparent PNG for the background of our page heading. It gives the heading a little added contrast, and provides a neat effect.

The example looks like Figure 2.

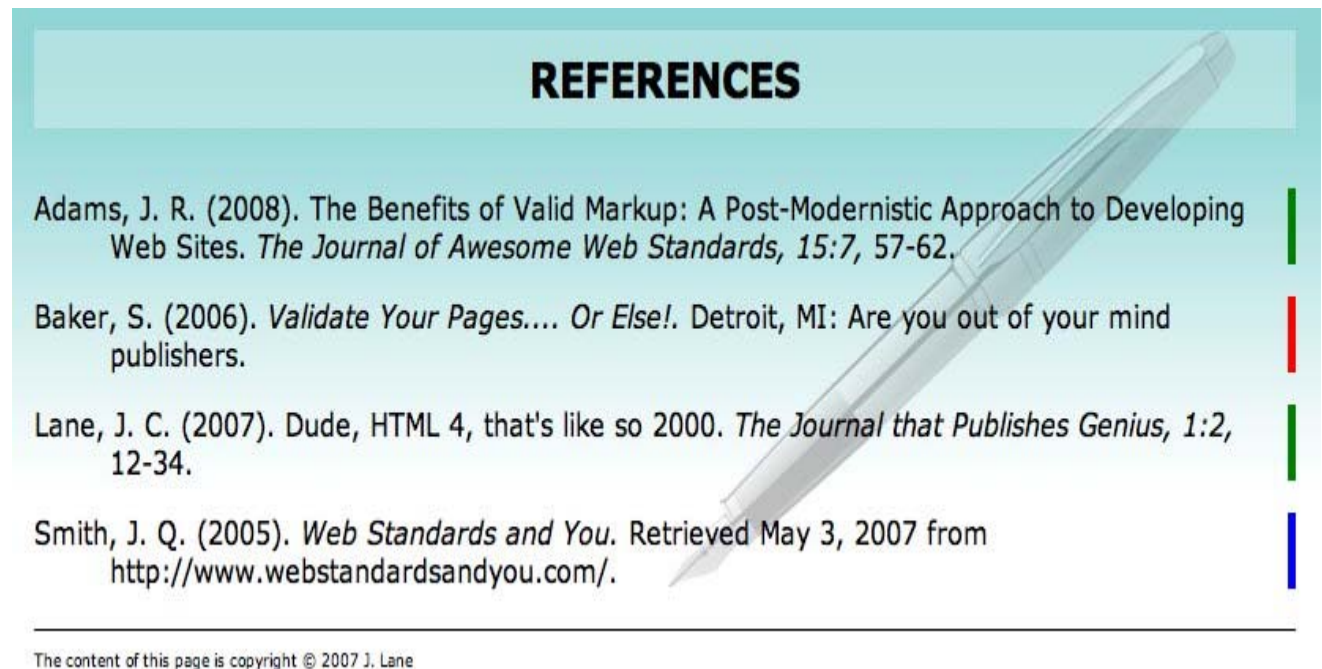


Figure 2: The finished example with styles applied.

Summary

There is nothing mystical about XHTML, CSS and JavaScript. They're simply an evolution of the web. If you've already had some exposure to HTML, there is nothing to "unlearn". Everything you know is still relevant, you'll just have to handle certain things a different way (and be a little more careful in your markup).

Aside from getting the satisfaction of a job well done, web standards development just makes sense. The counter-arguments to developing with standards is that it's time consuming and a pain in the neck having to make a layout work across browsers. The opposite argument could be applied to making a non-standards-based layout work across a range of devices and with future browser versions. How can you be

certain that your hobbled-together markup will display at all in Opera 12.0, Firefox 5.0 and IE 10.0? You can't, unless you follow some rules.

Exercise questions

- What's the difference between a class and an ID?
- What role do XHTML, CSS and JavaScript each play on a web site?
- Take the index.html file from the example provided, and change the formatting using the CSS alone (I'd suggest editing the file using a text editor such as Notepad or Text Wrangler). Do not make any changes to the HTML.
 - Add an icon for each of the different reference types (a different icon for articles, books and web resources). Create your own icons for this purpose, and make them appear alongside the different reference types using CSS alone.
 - Hide the copyright notice at the bottom of the page.
 - Change the look of the title, make it distinctive.
- What sorts of things could you do to the CSS to make this example work well on a mobile phone browser?

About the author



Jonathan Lane is the President of [Industry Interactive](#)—a web development/web application development company located on Mayne Island, British Columbia, Canada. He got his start in development working for the University of Lethbridge Curriculum Re-Development Center as their web projects coordinator for many years.

He blogs at [Flyingtroll](#) and is currently developing [Mailmanagr](#), an e-mail interface for the Basecamp project management application.

5: Web standards - beautiful dream, but what's the reality?

BY [JONATHAN LANE](#) · 8 JUL, 2008

Introduction

Up until this point, we've talked about the beautiful ideal of web standards. Web standards allow for interoperability between all web browsers, on every operating system, and even on every electronic device available. But is that really reality? Are all web browsers 100% standards-compliant? Are all web developers using web standards properly? Do web developers build a page using web standards, and then just walk away confident that their design will hold up everywhere?

The really simple answer to that last question is no; while that's an ideal situation, that is far from reality. In this article, I'm going to look at the following:

- [How do you check for web standards compliance?](#)
- [Standards compliance on sites today](#)
 - [Amazon: Shopping with standards?](#)
 - [CNN: Standardized news?](#)
 - [Apple: The pinnacle of elegance in design ... and validation?](#)
 - [A small standards compliance survey](#)
- [Why the lack of compliant sites?](#)
 - [Education](#)
 - [Business reasons](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

How do you check for standards compliance?

Before we go any further, the question you're probably asking yourself is "how do you know if a web site uses web standards?" Does it look any different to any other web site?

Yes and no. Web standards-compliant web sites, if developed properly, should look no different from web sites coded using a jumble of hobbled-together markup. However, the source code of the web site (try right/Ctrl-clicking on a web site and selecting the "Source" or "View Source" option—the exact terminology varies between browsers) will look vastly different. A standards-compliant web site will have nice, clean markup with little or no formatting embedded in the page itself. It might be hard for you to notice this at a glance, but trust me, visually impaired individuals using screen readers and search engines will notice right away. We have already covered the advantages of using web standards in the last article.

The easiest way to check for standards compliance is to use a handy tool, available online, called a validator. The World Wide Web Consortium (W3C) makes a validator freely available at <http://validator.w3.org/>—see Figure 1. You can (and should) use this tool to check any web sites you're developing for errors in your HTML/XHTML. CSS can be checked out using the CSS validator available at <http://jigsaw.w3.org/css-validator/>. Feel free to click through these links, and test a few of your favorite web sites.



Markup Validation Service

Check the markup (HTML, XHTML, ...) of Web documents

Jump To: [Congratulations](#) · [Icons](#)

This Page Is Valid XHTML 1.0 Strict!

Result:	Passed validation	
Address :	<input type="text" value="http://www.opera.com/"/>	
Encoding :	utf-8	<input type="button" value="(detect automatically)"/>
Doctype :	XHTML 1.0 Strict	<input type="button" value="(detect automatically)"/>
Root Element:	html	
Root Namespace:	http://www.w3.org/1999/xhtml	

Options

☐ Show Source ☐ Show Outline ☒ List Messages Sequentially ☐ Group Error Messages by type

☐ Validate error pages ☐ Verbose Output ☐ Clean up Markup with HTML Tidy

[Help](#) on the options is available.

Figure 1: The W3C’s markup validation service will check your pages for you and point out any errors in the markup.

Ensuring your pages validate is only half the battle. How do we check if browsers are standards-compliant? The Web Standards Project has developed a series of tests called the Acid tests, which use some complex HTML and CSS rules (plus some other markup and code) to see if a browser can render various test screens properly. The latest version of the Acid test, Acid3, is still a work in progress. You can read more about the Acid tests at <http://www.acidtests.org/>, as well as go to the actual test pages to put your browser through its paces.

Standards compliance on sites today

Are major web sites using web standards, or are they just hacking something together? Let’s take a look at a few different companies out there and see how they score using the W3C’s markup validation service. You’ll be surprised how many large web site don’t pass standards validation tests; don’t get disheartened however—there is no reason why you can’t go one better and get your sites validating properly. Bear in mind as you read the below reports that the larger and more complicated a web site is, the harder it is to make it validate, generally speaking (there are other factors to consider, such as the technologies used).

Amazon: Shopping with standards?

Chances are that if you’ve ever done any online shopping, you’ve probably visited [Amazon.com](http://www.amazon.com) (or one of its country-specific web sites). Amazon is a megastore in cyberspace, offering everything from books to CDs to groceries in certain areas. Does Amazon.com validate though? Check out Figure 2.

This page is not Valid (no Doctype found)!	
Result:	Failed validation, 1500 Errors
Address :	<input type="text" value="http://www.amazon.com/"/>
Encoding :	iso-8859-1 <input type="button" value="(detect automatically)"/>
Doctype :	(no Doctype found) <input type="button" value="(detect automatically)"/>
Root Element:	html

Options

☐ Show Source ☐ Show Outline ☒ List Messages Sequentially ☐ Group Error Messages by type

☐ Validate error pages ☐ Verbose Output ☐ Clean up Markup with HTML Tidy


[Help](#) on the options is available.

Figure 2: Amazon.com fails with flying colors! Not only is there invalid markup, but they don’t even declare a doctype (saying what version of HTML/XHTML they’re using).

Amazon has a bit of a journey when it comes to standards-compliance. I don’t have the inside track at Amazon, but if I’m allowed to speculate for a minute, I’d say that because Amazon has been around for quite some time, they have probably been using the same software to power their web site for their entire lifespan. Because web standards didn’t really grab the spotlight until early in this millennium, chances are that the system that Amazon uses for selling products online was developed back when web standards weren’t really supported or publicized. I don’t know for sure, but I’d guess that Amazon suffers from a case of just sticking with what works for them.

CNN: Standardized news?


Surely news organizations are semantic beings? [CNN.com](http://www.cnn.com) is one of the largest media web sites around. With global resources, reporting on news stories as they happen, surely they’ve got a team of in-house specialists ensuring that their web site is produced with valid markup? Check out Figure 3.

 **Line 569, Column 23: required attribute "ACTION" not specified.**

`<form class="cnnHidden" >`

The attribute given above is required for an element that you've used, but you have omitted it. For instance, in most HTML and XHTML document types the "type" attribute is required on the "script" element and the "alt" attribute is required for the "img" element.


Typical values for type are type="text/css" for <style> and type="text/javascript" for <script>.

 **Line 610, Column 1472: end tag for "UL" which is not finished.**

`...div><div class="cnnSvcBull"></div><div class="cnnSvcMore"><a href=`

Most likely, you nested tags and closed them in the wrong order. For example `<p>...</p>` is not acceptable, as `` must be closed before `<p>`. Acceptable nesting is: `<p>...</p>`

Another possibility is that you used an element which requires a child element that you did not include. Hence the parent element is "not finished", not complete. For instance, in HTML the `<head>` element must contain a `<title>` child element, lists (ul, ol, dl) require list items (li, or dt, dd), and so on.

 **Line 626, Column 13: end tag for "DIV" omitted, but its declaration does not permit this.**

`</div></t><td class="cnnPLDivCell"><img src="http://i2.cdn.turner.com/cnn/im`

- You forgot to close a tag, or

Figure 3: CNN.com (as of April 15, 2008) fails validation with 33 errors. They list an HTML 4.01 Transitional doctype, but a lot of their markup looks a little XHTML-esque.

33 errors isn't that bad when it comes to a web site of CNN's size and complexity. Those 33 errors could (and again I'm speculating here) occur because the content management system they're using isn't completely up to par on producing standards-compliant markup, or it could just be a collection of markup errors by a production staff that specializes in writing the news, and not in producing web sites; either explanation is plausible.

Apple: The pinnacle of elegance in design ... and validation?

Apple is renowned for their beautiful and functional hardware and software products. Their product announcements are almost like religious experiences for droves of loyal followers. [Apple's web site](#) (see Figure 4) is often acclaimed as being beautifully designed and well organized, but does it validate?



Figure 4: Apple.com comes really close to having valid HTML 4.01 Transitional markup. With 6 errors, there's a mixture of markup "typos" and a case of Safari-specific tag use.

The Apple web site comes really close to validating. Really, it would take someone all of 5 minutes to fix the errors and have the page pass. The one error I want to briefly mention, however, is that Apple has decided to use a Safari-specific attribute on their search box (giving the search box the attribute `type="search"`). In Safari, this will let you see a list of recent searches by clicking on a small magnifying glass icon. In other browsers, like Opera or Internet Explorer, it just shows up as a normal text box.

A small standards compliance survey

Instead of going through dozens of examples like this, I've compressed the remaining sites surveyed down into a handy pie chart. I looked at about 40 corporate web sites from the Fortune 500 list as well as the Alexa rankings of web sites with the most traffic—Figure 5 shows what I found:



Figure 5: 85% of web sites surveyed failed validation on some level. Some were spectacular failures of upwards of 1,000 errors; others were just a couple of typos here and there.

Why the lack of compliant sites?

We're left crying: "why, why can't they just validate?" That may be a little dramatic, but it's at least similar in content to the question running through your mind at this point. Why do so few web sites validate? I've talked about a few possible reasons already—things like legacy e-commerce systems or content management systems—but there's a few other underlying reasons as well.

Education

The school I went to had a Management Information Systems program, a Computer Science program, and a New Media program, each of which had courses related to web site production—while these taught many things effectively, there wasn't really much coverage of how to actually code a web site in any of them. The general feeling I get from looking at numerous university courses is that web languages like HTML, CSS and JavaScript are below the technical threshold of most computer science programs, and above the technical threshold of most MIS/New Media programs.

What I'm getting at here is that many educational courses don't cover this kind of stuff in any great level of detail. I would be willing to wager that if you ask 10 developers who work with web standards where they learned how to use web standards that 9 of them would reply that they are self-taught (the other 1 won't answer you because she's too busy trying to get her site to render properly in IE6).

The World Wide Web Consortium (W3C), which is the group responsible for developing standards, and the Web Standards Project (WaSP) are taking on this challenge though and are really pushing to have web standards support improve, both from browser manufacturers and from developers.

Hell, the entire reason this course you're currently reading was created is to provide a proper set of teaching material for web standards, and a means by which to use that material to learn, for free. We're just trying to get rid of a few more of the reasons (we hesitate to use the word "excuses" here...) why people aren't adopting web standards. There's really no excuse for not using them, given the benefits they incur (as discussed in the last article).

Business reasons

A web site that I frequently visit for entrepreneurs involved in Web-based startups has hosted a number of discussions about the use of web standards in "Web 2.0 applications". There is usually an interesting exchange between those who believe that web standards should be used because they make sense (for all of the reasons we've previously discussed), and those who just say "so what".

The fact of the matter is that web browsers will handle really bad code. Your pages don't need to validate in order to have them display properly in most of the major browsers. From a business perspective, where time equals money, why bother spending any additional time to get them to validate at all? If you can

crank out a table-based mess of code in 30 minutes, or spend 30 minutes coding your page in HTML and CSS and 30 minutes making sure that it validates and works ok across browsers, and the end result will look the same in the major desktop browsers, which sounds easier to you?

A lot of people from my generation (I’m almost at the big 3-0, at the time of this writing) learned to build web sites using tables for layout, and font tags to deal with typography. It can be intimidating to re-learn how to do something when what you’re doing still “works” (still looks fine in most web browsers). Employers generally don’t know the difference; I’ve never once had a manager talk about the quality of my markup during a performance review. So really, where’s the incentive?

I’m going to throw in here (you can guess which side I’m usually arguing) that taking the messy-code approach is shortsighted. Based on my experience, doing a re-design of a standards-based web site is much easier than converting a mess of improperly coded pages (I’ve done both). I have yet to hit that utopia promised by XHTML/CSS that you will only have to touch the CSS during a re-design, but I’ve come close.

Also bear in mind that you will see a lot more web job adverts these days asking for knowledge of web standards than ever before.

There are also some direct business reasons to consider. In general, using web standards improves a web site’s position in search engine rankings (how high up it appears on the list when you do a Google search) and how easy it is to find in the first place. In addition, using standards and best practices will generally make the web page more accessible by people with disabilities trying to make use of it, and users trying to access the site on their mobile phones. Extra users and increased visibility is always good for business.

Summary

In this article I’ve talked about the state of web standards adoption today—how to check whether standards are being used properly on a site, how many sites are using web standards properly and the reasons why people don’t adopt standards. As you’ve seen above, the reasons aren’t really that compelling and should be fairly easy to overcome.

So what’s an enterprising web developer of the future to do? Do you bother with web standards (and keep on reading this series), or do you fire up a graphical editor and start laying out your web site in tables?

Let me put it this way: the single biggest complaint I’ve read from individuals who say that standards-based development is a waste of time is that it takes too long to learn to use web standards instead of outdated methods and develop web sites that work across all browsers. So why not start out by learning the correct way to do it, and save yourself some trouble? You’ve decided to learn how to build web sites, and you need to do it one way or another, so why not learn how to do it properly?

Further reading

- [W3C markup validation service](#).
- [W3C web site](#) (with information about various standards and future recommendations).
- [The Web Standards Project](#).

Exercise questions

- We looked at a lot of “big” web sites and whether they validate or not. Run a few of the sites that you visit regularly through the validator. Do they validate? If not, take a look at some of the errors to get a sense of why they’re failing.
- What is a doctype? What does it do?
- What case can you make for web standards as it relates to business?

About the author



Jonathan Lane is the President of [Industry Interactive](#)—a web development/web application development company located on Mayne Island, British Columbia, Canada. He got his start in development working for the University of Lethbridge Curriculum Re-Development Center as their web projects coordinator for many years.

He blogs at [Flyingtroll](#) and is currently developing [Mailmanagr](#), an e-mail interface for the Basecamp project management application.

6: Information Architecture - Planning out a web site

BY [JONATHAN LANE](#) · 8 JUL, 2008

Introduction

Traditionally, the planning stage of a web site (or any project) can be a little stressful. Everyone has an opinion about how a web site should be built, and often their opinions will conflict with one another. Your number one goal on any web site should be to build something that's useful for the people who will be using it. It really doesn't matter what your boss says, what that guy down the hall with a doctorate in software engineering says, or even what your personal preferences are; at the end of the day, if you're building a web site for a particular group of people, their opinion is the only one that matters.

This article is going to look at the early stages of planning out a web site, and a discipline that is commonly referred to as Information architecture, or IA. This involves thinking about who your target audience will be, what information and services they need from a web site, and how you should structure it to provide that for them. You'll look at the entire body of information that needs to go on the site and think about how to break that down into chunks, and how those chunks should relate to one another. The sections below are as follows:

- [You need to plan out the site you're building](#)
 - [Introducing "The Dung Beatles"](#)
 - [Now what? Drawing a site map](#)
 - [Naming your pages](#)
 - [Adding some details](#)
- [Summary](#)
- [Exercise questions](#)

You need to plan out the site you're building

You'll come upon the odd web project that you can just dive right into without any up front thought, but these are, by far, the exception and not the norm. We're going to take a look at a fictional band called "The Dung Beatles" and try to help them work through the early stages of planning out their web site. We'll talk with the band and find out what goals they have, and what they would like to see on their web site. Then we'll dive in and start working on a structure for the band's information.

Introducing "The Dung Beatles"

The Dung Beatles (TDB) have a problem. They are the hottest Beatles tribute band in Moose Jaw, Saskatchewan, but they need to raise their profile for an upcoming North American tour this summer. They've got venues scheduled throughout Canada and the United States, but they're virtually unknown outside of their hometown. If only there was some way, using technology, to reach a large number of Beatles fans for relatively little money.

Lucky for TDB, we've got this thing called the World Wide Web, and they quickly decide that building a web site is the answer they've been searching for. TDB needs a place to promote their tour dates, build a fan base in other cities and raise awareness of the band. You're going to work through their ideas with them and see if you can chart out a plan for their web site.

You schedule a meeting with your new clients to hash out the details of what they're looking for and to decide on due dates and costs. You open the conversation by suggesting that you talk about the goals and objectives of the web site in order to get an idea of what they want. What does the band hope to achieve with their online presence?

TDB starts talking about their upcoming tour, and how they want to get the word out to Beatles fans in all of their scheduled stops. It's February now, and they're scheduled to kick off their tour in five months time.

Hang on a second! A web site alone won't build it's own traffic and publicise itself. You extract from the conversation thus far that the main goal for the site is to provide a home for TDB fans online; a place where they can keep up to date on the latest news, tour dates and venues. Through the fans (word of mouth), and some other advertising venues, new people will be driven to the web site where they can download sample tracks, check out pictures of the band (in full costume) and find out where/when they can check them out live.

Raul McCoffee, the front man of the group, points out that it would be nice to be able to raise a little extra money for the tour through the sale of some CDs and band merchandise. You gather the band around and draw out a quick sketch of what a visitor might want when they visit the web site. This is just a really rough brainstorm of ideas; it's got very little structure at this point.

There are two general groups of people who will visit the site—people who know TDB already and like them (fans), and people who are unsure. You've got to cater to both those groups in different ways; potential fans need to be “sold” on the group, whereas current fans want to “feed their addiction” (so to speak). What sort of information is each of these groups going to be looking for? Figure 1 gives an indication of this—this is a typical sketch of the type that you'll want to make at this point in future web site projects. From this, you'll work out what pages the web site needs, and how they should link to one another.



Figure 1: What your web site visitors want.

You settle on a budget, and agree to launch the web site in one month. You promise to get back to the band in a couple of days with some plans outlining the direction you're going in.

Now what? Drawing a site map

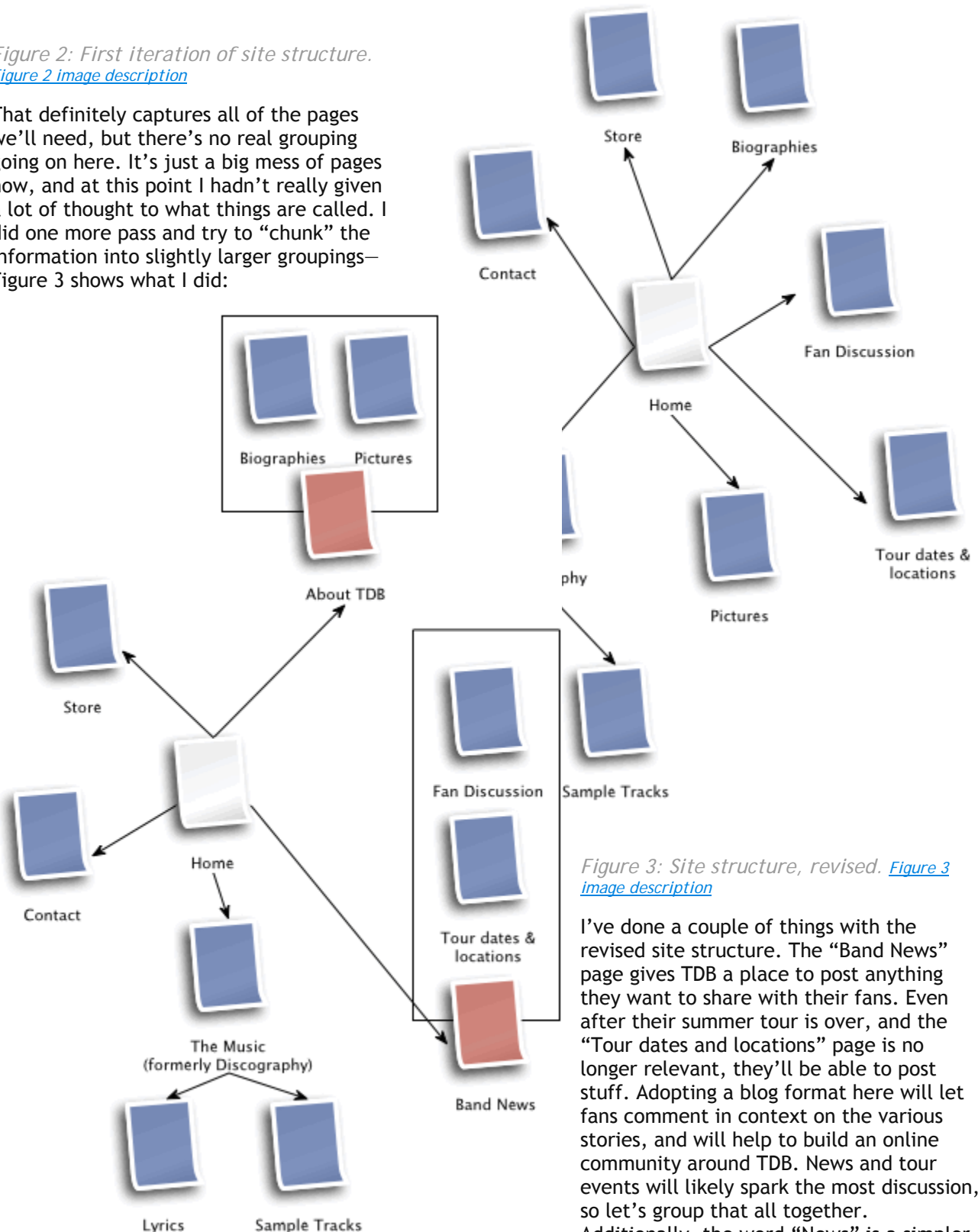
A lot of people will throw together a site map at this stage—this looks like an org (organisational) chart. This is usually a pretty basic graphic showing simply the names of each page on the site and how they link into the overall structure of the web site. Personally, I like to put in a little more detail and talk about the purpose and content of each page. For example, a page may be labeled “Home”, but what is the home page? Is it a cheesy “welcome to our web site” message (yuck!) or is it a more dynamic page containing news items and enticing images? Take a few minutes to think about what pages the above sketch might turn into, and what might be contained on each page. Have a go at drawing your own site map before moving on to the next section.

Now let's get started with the basics: one of those org charts that I mentioned above. Figure 2 shows my attempt at taking the brainstorm and turning it into a site org chart:

Figure 2: First iteration of site structure.

[Figure 2 image description](#)

That definitely captures all of the pages we'll need, but there's no real grouping going on here. It's just a big mess of pages now, and at this point I hadn't really given a lot of thought to what things are called. I did one more pass and try to "chunk" the information into slightly larger groupings—Figure 3 shows what I did:

Figure 3: Site structure, revised. [Figure 3 image description](#)

I've done a couple of things with the revised site structure. The "Band News" page gives TDB a place to post anything they want to share with their fans. Even after their summer tour is over, and the "Tour dates and locations" page is no longer relevant, they'll be able to post stuff. Adopting a blog format here will let fans comment in context on the various stories, and will help to build an online community around TDB. News and tour events will likely spark the most discussion, so let's group that all together.

Additionally, the word "News" is a simpler, more general word that people will be able to recognise faster if they're skimming a page for the information they want.

Our new “About The Dung Beatles” page groups together the band members’ biographies as well as their pictures. Going this route gives us a jumping off point for individual band member biographies. Following a similar argument to the one we made above, “About” is a common term used on a lot of web sites. Any time a visitor wants to learn more about a company, a product, a service, or an individual, they usually look for an “About” link.

Finally, the term “Discography” is a bit of a technical term. It’s possible that fewer people will understand what that term means than “The Music”. Also, it opens up this page to additional content: sources of inspiration, history of a particular song...you get the idea. I think we’re ready to roll. After I’ve talked a bit about naming pages sensibly, we’ll move on to add a little more detail about each page.

Naming your pages

Page names can be one of the most crucial decisions you’ll make during web site design. Not only is it important for your visitors so that they can find their way around your web site; it is also another thing that dictates how easy your site is to find using a search engine (you’ll find various mentions of search engine optimisation throughout the course).

In general, search engines look at the text included in a web page, the URL of that page, and the text of any links to that page when they’re deciding “how important” it is. Giving your pages sensible names and sensible URLs will encourage anyone linking to your pages to use sensible descriptions.

Here’s an example. Let’s say you’re a car company, and you have a model called “The Speedster”. You’ve got a web site to promote your automobile, and one of the pages lists available features. Do you call this page “Features”, “Available Features”, “Features of the Speedster”, or “Bells and Whistles”? I would suggest that “Features of the Speedster” is the best option from this list. It’s specific to what the page contains, chances are that the title will be displayed high up on the page and will be prominent (good for search engine indexing), and you may even be able to fit it into the URL (something like “www.autocompany.com/speedster/speedster-features/”).

Adding some details

You don’t have to figure out everything at this point, but you need to at least provide a brief description of what you have in mind for each page. After you’ve got the site structure, number each of your pages and provide a brief description for each page, like I’ve done in Figure 4 for the home page (you’ll get a chance to do this for the other pages in one of the exercises questions at the end of the article.)

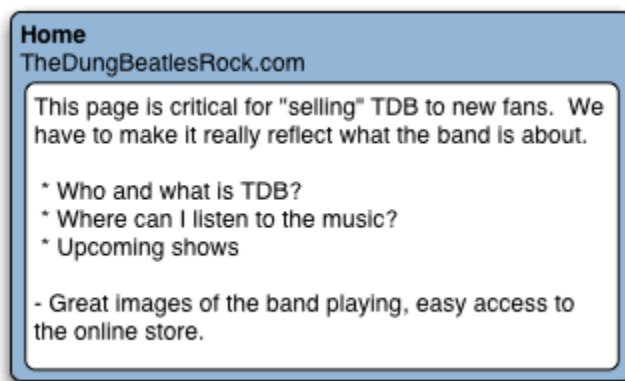


Figure 4: Page Details for the Home page. [Figure 4 image description](#)

This is about as involved as you want to get at this point. You don’t need to describe page functionality, the technology you’ll use to build it, or the design/layout in great detail. Just describe what you have in mind in general terms. Your goal here is to communicate what you’re thinking to your client and to force you to think things through.

It’s not uncommon at this stage to come to the realisation that you have too many pages, and you’ll never be able to find content for them. You can go crazy in creating a hierarchy of pages. For example, if the band members just wanted to publish one paragraph about themselves, it wouldn’t be necessary to create separate biography pages for each member. They could all be combined into a single page.

Summary

This article has looked at the web site as a whole, and how you should think about structuring it. In the next article, you're going to get taken down to the page level, and look at what goes into making a great web site: what features to include and where to include them. Articles 8, 9 and 10 then look at the visual design of a page. So this is being done in 3 logical steps (check it with the client at each stage to make sure they are happy with it):

1. First you decide on the content of a web site, and decide how to structure that content into pages.
2. Next you decide on the functionality that will actually be used on your web site.
3. The last thing you do before you actually start going ahead and coding your web site is work out the visual design of it—the page layouts, and the colour scheme, etc.

Exercise questions

- Look back at Figure 1 and try to develop a similar brainstorm for a web site about a car (pick any current or imaginary car).
 - What will visitors to the web site want to know?
 - Is there anything at existing car web sites that you see as essential? Frivolous?
- Take your brainstorm and try to organise the information. What page groupings make sense?
- Another activity that is sometimes useful when planning out a web site is to check out the competition. Do a search for band web sites (bonus points for tribute bands), and take a look at what they're offering. Did we miss anything?
- Take a look at Figure 4 and try to develop similar figures for the other pages I've identified on the web site.

About the author



Jonathan Lane is the President of [Industry Interactive](#)—a web development/web application development company located on Mayne Island, British Columbia, Canada. He got his start in development working for the University of Lethbridge Curriculum Re-Development Center as their web projects coordinator for many years.

He blogs at [Flyingtroll](#) and is currently developing [Mailmanagr](#), an e-mail interface for the Basecamp project management application.

7: What does a good web page need?

BY [MNFRANCIS](#) · 8 JUL, 2008

Introduction

Continuing on from [the previous article](#), in this article I will look at the content of The Dung Beatles' web site to give you a flavour of what good web sites and pages need to contain.

You won't be looking into the actual underlying code yet; instead you'll just start to examine the different pages, thinking about what items should appear on them, and considering issues such as consistency, usability, and accessibility. The table of contents is as follows:

- [The home page](#)
 - [What does this mean for our site?](#)
- [Navigation](#)
- [Other common elements on a site](#)
- [Context is everything](#)
 - [Relevant content](#)
 - [Headings](#)
- [Usability](#)
- [Accessibility](#)
- [Summary](#)
- [Exercise questions](#)

The home page

At this point, many people tend to think “let's start at the first page many users are going to see—the home page. That's logical isn't it?”

It may sound logical, but then again, it's not really the best place to start. It can be a common mistake to focus upon the home page. Home pages of websites can become hot property, trying to summarise everything in the site, and be everything to everyone. This can lead to them becoming a bloated mess.

As an example of what I mean, take a quick moment to visit the [MSN home page](#) (see also [Figure 1](#)). Marvel at the vast over-abundance of links and content. The MSN network of sites is vast—from travel to TV, dating to directions, gadgets to green information. And all of them vying for your attention.



Figure 1: The MSN home page—that's a lotta links!

This kind of overwhelming “throw everything in but the kitchen sink, then add the kitchen sink” home page is probably fine for such a large collection of sites, but for our band home page it is definitely over-kill and would turn away many more users than it would attract.

Also, it is a common misconception to think that the home page is the first page your site visitors will see. Maybe if they hear about the band or get a flyer or sticker or badge/button about it, they will enter the site's location into their browser and so end up on the home page.

What is more likely however is that your visitors will end up on your site based upon a search. If they search for the band name it is likely (although not guaranteed) that they will have the home page of the site as their top result. Say for example someone searches for “Beatles tribute gig” instead though—they may well get the “Tour Dates” page as the first result. Or if the search was “band Moose Jaw” the top result may be the “About TDB” page because that mentions that the band is from Moose Jaw, whereas the home page does not.

In an [article about their decision to stop charging for access to old content](#), the New York Times notes that the behaviour of their visitors had changed—What changed, The Times said, was that

...many more readers started coming to the site from search engines and links on other sites instead of coming directly to NYTimes.com. These indirect readers, unable to get access to articles behind the pay wall and less likely to pay subscription fees than the more loyal direct users, were seen as opportunities for more page views and increased advertising revenue.

What does this mean for our site?

All of this means that whilst you need to sub-divide your content up into individual pages, you should be thinking about how your visitors will find what they might really be looking for, or where in the site they might go next once they start exploring further.

Whilst it may be tempting to cram in a little of everything on the home page, it is actually better to use it to highlight other areas of content within the site and drive your traffic to them. Treat the home page as any other page on the site and give it a defined purpose (to show what's new, to provide an overview, to just introduce the band and lead people further into the site, and so on). The page will also need some form of navigation to other areas of the site, and branding.

Now you will go a bit deeper, and learn more about these things...

Navigation

How to navigate a site is one of the most, if not the most, crucial aspects of implementation to get right. You should identify the common destinations of your site and put these into your main navigation.

There is another common misconception about navigating sites that you may have heard—that any page should be at most three clicks away. The spreading of this has led to some of the worst and most complex navigation on the internet. As a real-life example, look at most shopping or price comparison sites on the internet—they tend to try to cram in as many links as possible for the mantra of keeping their customers' clicks as few as possible before they purchase something, lest they leave and use a competitor. Eventually, though, this will lead to too much information for the user to take in and use effectively. Too much choice is as paralysing as too little choice.

As long as there is an apparent progress from one link to the next, and the indication is that they are still on track to reach their end goal, users will tend to continue through a site.

Taking the cues from the IA structure, the main navigation of TDB's site should contain links to the pages/sections “Store”, “About”, Contact, “The Music” and “Band News”, as well as a link back to the Home page. It is not really necessary to link to pages below that, such as the “Tour Dates” and “Lyrics”

pages. Links to these pages should be found within that area alone—anyone needing to jump from a particular song lyric page directly to “Tour Dates” will be able to navigate to “Band News” and hence to “Tour Dates”.

One of the most crucial aspects of a successful site navigation is consistency. Take a look at the tabs at the top of this page (links such as “Home”, “Articles”, “Forums” and so on). As you move around the dev.opera site this navigation bar will remain. The navigation will change to indicate where you currently are in the site and provide further links to deeper within the area. For example, clicking on the “Articles” tab will take you to the main articles area, which then contains links to some of the most recent articles and a collection of links to sub-areas on topics like “Accessibility”, “CSS” and “Mobile” (see also Figure 2).



Figure 2: The dev.opera.com navigation is constant wherever you are on the site.

Other common elements on a site

Apart from the navigation, there are normally other parts of a site that appear on pages repeatedly.

Most sites will have some form of branding, logo or masthead to show ownership. For example, almost every page on a Yahoo! site will have the logo in the top-left position, and this logo should contain the part of the Yahoo! network you are on (such as “Travel”, “Movies”, “Autos”, etc).

The header (across the top of the page) can contain more than just a logo, however. It can also contain, or be attached to, the main navigation. A search box is not uncommon, allowing users to search through the site rather than navigate around using menus and links. You should be including most or all of these elements on each page of your site.

The footer (the last area of the page) should contain extra information such as your copyright notice and links to useful ancillary pages on your site if you have them (such as “About This Site”, “Terms & Conditions”, “Contact Us” etc).

Colour, layout, the use of shapes and icons, typography and imagery all combine to create an overall impression of a page as ‘belonging’ to the site it is on—consistency is key. The use of consistent appearances and placement help to keep you oriented as to where in a site you’ve ended up and creates a sense of familiarity. You know that the page you are on now is related to and a part of the same experience as the previous pages because they are visually connected and related. When you design the site you must bear this in mind and not create a different look for all of the pages in the site.

In our TDB site, the header of the page would contain the band logo and name to reinforce this to visitors as they move around the site and give them a sense of familiarity that they are still looking at information about the band. The footer would contain copyright information about the site and about the lyrics, images and audio samples on the site; it should also contain links to pages on contacting/booking the band.

Context is everything

Each page, despite all of the common elements, should be unique. A good web page will do one thing, or a small number of things, and do them well.

Relevant content

Making content relevant and separate is a key factor of making good web pages great. Content should be uniquely addressable (have a place where it definitively lives, at a unique URL) and logically ordered (both within the site and the page itself) so that it can be easily found.

Take upcoming gigs performed by the band—whilst it could be tempting to place an “upcoming gigs” module on each page, that information has a home of its own and should live there. A simple “next gig” module that also links across to the Tour Dates page can promote that just as effectively without duplicating content and potentially confusing both users and search engines.

Headings

The next time you have your hands on a newspaper, really look at it. Notice how some stories are bigger, have darker type and imagery, and more prominent headlines. You are effectively being told what the most important stories to read are if you are in a hurry and only want to catch the big news.

The same is true of web pages. Each section of content on a page should be introduced by a heading, indicating it’s relative importance (is this section subordinate to the previous one, or on the same level?) within the page.

As an example, in the current part of this very article page are the two lines “Relevant Content” and “Headings”. These are headings and are at a lower level than “Context Is Everything” to indicate that these are sub-sections of the Context section of the page.

Usability

Usability is a catch-all term for making a site behave in a rational and expected way.

Have you ever tried to read an article on a news portal only to find you had to register with them before reading it? Ever tried to book a flight or a train journey online in under two minutes—the time you might expect it to take to explain the journey you wish to purchase tickets for to someone over the phone or over a counter? Ever entered an address or a credit card number only to be told you had entered it in the wrong form? Ever have a search return no useful results even though you know they are there, or for a site to not have a search facility in the first place?

These are all examples of bad usability—stemming from not considering the needs of the site’s users. By placing those needs at the centre of the experience you design and create, you are much more likely to create a satisfying and rewarding site.

Creating usable sites is not easy, however, and much of the knowledge simply comes from experience. Keep a diary of things that annoy you on other sites, and learn not to do them in your own. Also, nothing beats testing the site on real people. Once you’ve created it watch people as they use it:

- can they find pages they are looking for?
- does the search give them the right results for the search terms used?
- do images/audio/video work in their browser?
- do they get annoyed at anything?
- are they pleased by anything?

Dedicated usability testing is something professional companies will charge a lot of money for, but even some quick informal testing with friends and family can give you valuable insights into problems that you have not noticed. This is because you created the site, so can second guess things—other people, however, can’t do this.

Accessibility

Accessibility is, unfortunately, most commonly understood to mean “making a website work for blind people.” The truth is that accessibility issues affect many, many more people.

The term “assistive technology” is used to describe any extra piece of computer equipment or hardware that helps a person interact with their computer more successfully. This normally brings to mind screen readers for people who cannot see, or voice input for people who cannot use a mouse or keyboard. But—what about glasses? People who need to correct their vision are also using an assistive technology. But most would not classify themselves as being disabled.

An awareness of the problems that people can face when trying to use the internet is important when developing good web pages. Don't assume that your users will have a mouse; don't assume that people can see the images you have used; don't assume that everyone has Flash installed and provide alternative content for those without.

In addition to people who require assistive technology, there are other people these things are true for such as mobile users. Flash is still quite an immature technology for the mobile phones that even have it—the Apple iPhone doesn’t have it and possibly never will, but otherwise can browse the web almost like the desktop version of the Safari browser on a Mac (Opera Mobile does support Flash). Technologies like Opera Mini that allow lower powered mobile phones to access the web will rewrite web pages to be lighter and more efficient and imagery in pages may be rendered much smaller for most users—meaning any content relying on subtlety of detail may be missed.

In the case of our band, you should be aware that if you use a lot of imagery (band photos), you should be describing the content of the image. If you use a Flash music player in the page to allow people to listen to the band's music, you should also link to the music directly so that those without Flash installed can still access the music in their preferred way.

Summary

In this article I've discussed in broad terms what you will need to bear in mind as you start creating actual web pages, to make them more usable, more accessible, and run more smoothly. All of the concepts will be explained in greater detail as this series progresses.

Exercise questions

For the exercise questions for this article, just go and browse the web—visit some of your favourite sites, and look at them in a new light. Jot down some answers to the following questions:

- Do they have consistent headers, footers and navigation areas?
- Observe how the navigation changes as you move around the site.
- Pay attention to see if anything about the site annoys you or confuses you, and if so work out what the site *should* be doing at that point.
- If you can, turn off support for images or JavaScript in your browser, or use your mobile phone and compare the experience to using the same site on your computer.

About the author



Photo credit: [Andy Budd](#).

Mark Norman Francis has been working with the internet since before the web was invented. He currently works at Yahoo! as a Front End Architect for the world's biggest website, defining best practices, coding standards and quality in web development internationally.

Previous to Yahoo! he worked at Formula One Management, Purple Interactive and City University in various roles including web development, backend CGI programming and systems architecture. He pretends to blog at <http://marknormanfrancis.com/>.

8: Colour Theory

BY [LINDA](#) · 8 JUL, 2008

Introduction

Without much colour or graphics, every site would become [Jakob Nielsen](#)'s dream site. Although Nielsen's philosophy of bare-bones Web architecture has accessibility and usability merits, most Web designers want to add a signature touch to their sites with many design elements. Fortunately, a designer can add colour to a site without losing accessibility and usability if the site is designed with those capabilities in mind. While many designers feel comfortable designing a site for many users, those same designers might feel inadequate when it comes to choosing colours and graphics.

In this article, I'll cover colour basics and three simple colour schemes so that you can feel confident about choosing colours for your site. I'll follow up this article with another piece on how to simplify these colour choices. After all, it's more fun to enjoy the compliments on your Web site design than it is to sweat over the colour choices. The contents are as follows:

- [Colours, tints, and shades](#)
 - [Monochromatic colour schemes](#)
 - [Complementary colour schemes](#)
 - [Warm vs. cool colours](#)
 - [Triadic colour schemes](#)
 - [Tetradic colour schemes](#)
- [Summary](#)
- [Exercise questions](#)

Colours, tints, and shades

Colours, or hues, have historically been divided into primary, secondary, and tertiary colours. The primary colours consist of red, yellow and blue, and they're called primary colours because you don't need to mix colours to make these three hues. If you want to translate these colours into web colours, you can recreate them using their hex (hexadecimal) equivalents of #ff0000, #ffff00, and #0033cc as shown in Figure 1:

Figure 1: the primary and secondary colours, and their hex equivalents

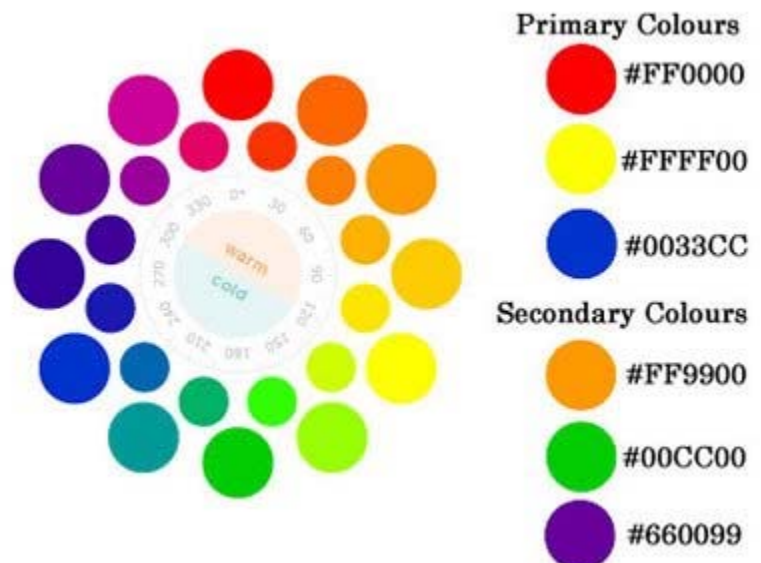
Secondary colours are mixed from primary colours, and those colours are as follows:

- Red + Yellow = Orange (#ff9900)
- Yellow + Blue = Green (#00cc00)
- Blue + Red = Violet (#660099)

Tertiary colours are mixed from the secondary colours, and they lie between the primary and secondary colours shown on the wheel above. Although web colours differ from regular

"painters" colours, it might help to [get](#)

[hold of a colour wheel](#) (as seen in Figure 2) to have at hand while you learn about various colour schemes. In addition, a colour wheel will show all the tints, tones and shades so you can begin to realize the colour possibilities you have at hand. Some more important terms to learn are as follows:



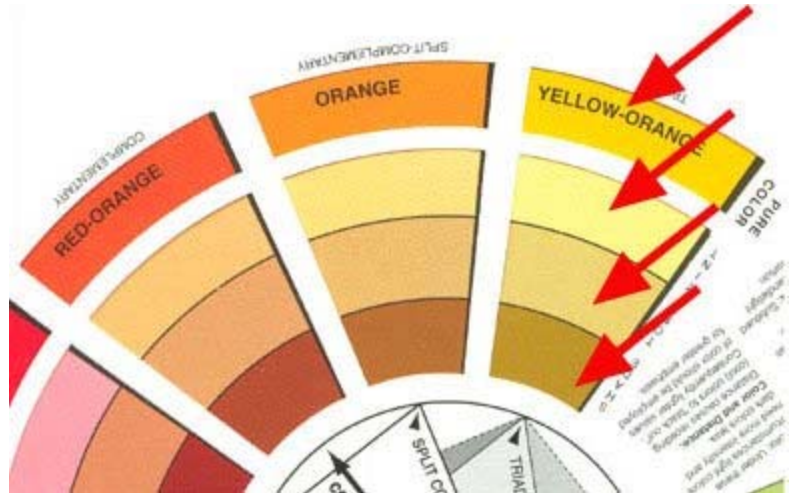
- Tint - The resulting colour when white is added
- Tone - The resulting colour when gray is added
- Shade - The resulting colour when black is added

Figure 2: A real colour wheel

The arrows in Figure 2 indicate different things as follows:

- Outermost band - tertiary colour of yellow-orange (yellow + orange)
- Second band - the tint of that tertiary colour (white added)
- Third band - the tone of the colour (gray added)
- Innermost band - shade on the print wheel (black added)

As you can see from the colour wheel shown above, the amount of white, gray and black added to a colour are minor—just enough to alter the original colour and to create what is known as a monochromatic colour scheme.

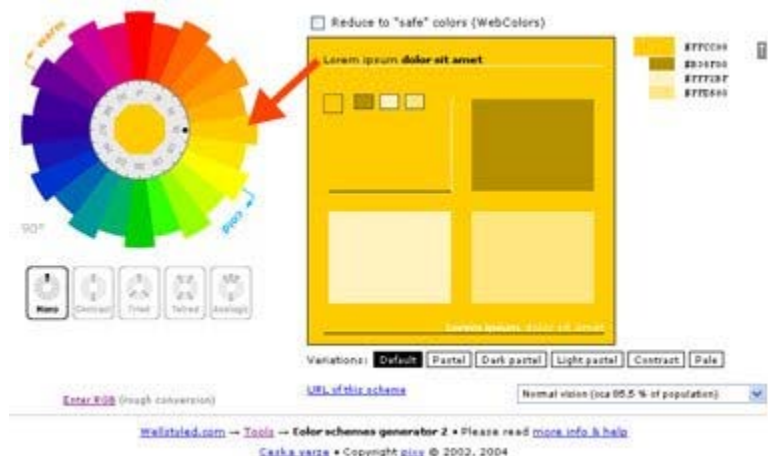


Monochromatic colour schemes

Colour schemes have been around for centuries, so there's no need to reinvent the colour wheel. Although web colour differs from print colour, the concepts are the same. You just exchange hex numbers for colour names, and match them as closely as possible. One online tool I suggest using to help out with this is the [Color Scheme Generator II](#), as seen in Figure 3, which allows you to determine colour schemes quickly and easily, and even determine whether the colours you've chosen provide enough contrast for low-vision or colour-blind users.

Figure 3: Color Scheme Generator II.

If you want more help deciding whether the colours you've chosen provide a good enough contrast, try out the [Contrast Analyser](#) from the Paciello Group. This tool checks the contrast between foreground and background colours.



To achieve the tint, tone and shade for the yellow-orange colour at the online colour generator, first select the colour that the arrow points to in the image shown above. Then, select Mono in the panel located under the colour wheel and Default in the panel in the box to the right. Also select Normal Vision in the selection from the pull-down menu at the bottom right. Do not check the "reduce to 'safe' colours" box above the colour box unless you're a purist.

Note: The term "[Web safe colours](#)" comes from the time when a computer monitor was capable of displaying only 216 colours. These 216 colours would appear the same across various computer platforms and browsers that contained a 256 colour (8-bit) computer system, so they were termed safe to use across platforms. While some purists still stick with the "Web-safe colour palette", modern browsers are capable

of handling what is known as "24-bit" colour. Actual 24-bit colour at ten to eleven bits per channel produces 16,777,216 distinct colours. In other words, it's safe to say that the "Web safe" colour palette rarely is needed anymore.

Back to the monochromatic colour scheme. The results you should receive by following the steps described above are: yellow-orange (#FFCC00), tint (#FFF2BF), tone (#FFE680), and shade (#B38F00). These hex numbers are much more reliable than any guesses you will make by trying to match a tangible colour wheel to the backlit screen of a Web browser. And, as the "Mono" suggests, this scheme translates to a monochromatic colour scheme, as seen in Figure 4.



Figure 4: A monochromatic colour scheme.

A monochromatic colour scheme equates to one colour and all its tints, tones and shades. While this scheme is the easiest to use, it doesn't provide much excitement in a Web design for many designers. Instead, you may want to explore other schemes to add pizzaz to your links, images, and banners.

Complementary Colour Schemes

The next colour scheme family to explore is the complementary scheme, where you match up colours that lie directly opposite each other on the colour wheel, as seen in Figure 5.

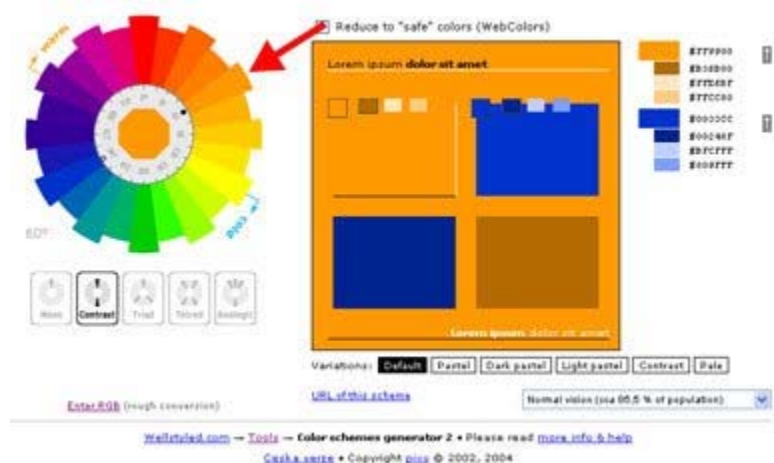


Figure 5: Examples of complementary colour schemes.

When you choose one colour and its opposite colour, you also include all the tints, tones and shades of both colours. This provides a wider range of choices, and it translates well with the online colour tool—see Figure 6.

Figure 6: A complementary colour scheme inside the online colour tool.

In the image above, I have chosen an orange colour with the opposite complementary colour of blue. The settings I chose to get to this scheme include the Contrast setting at lower left, the default on the menu below the generator, and normal vision. Notice that the main colour selected is marked by a black dot on the inner disc of the colour wheel (both above and online at the generator site) and that the opposite, complementary colour that was automatically picked for you is marked with a hollow circle in the inner rim. These markings make it easier for you to analyze your colour scheme.



This colour generator makes it easy for you to choose colours for links, visited links, and even images as it provides the hex colours for you at the upper right. You can mix and match any pure colour (the colour at the top) and its tint, tone or shade and feel great about choosing a solid colour scheme.



Figure 7: The Greenpeace site—a good example of a complementary colour scheme.

[Greenpeace USA](#) (see Figure 7) is one of many sites that uses a contrasting colour scheme. Yes, you see yellows and oranges, but the predominant colours are green and red—two colours that are directly opposite each other on the colour wheel. You almost can't go wrong with this complementary colour method. In fact, the use of a "warm" and "cool" colour combination makes a site zing with colour contrast.

Warm vs. cool colours

Complementary colour schemes are great to use in web sites, as they also contain both warm and cool colours. The use of these colours provides contrast, and it's easy to remember which colours are "warm" and which colours are "cool" as you can see in Figure 8 (and on the colour generator site):

Figure 8: Warm and cool colours.

Warm colours are those colours that would remind you of the summer, sun or fire. They consist of violets to yellows. Cool colours might remind you of spring, ice, or water. Those colours range from yellow-green back to violet. If you notice how the colours work on the wheel, you'll soon discover that you can't choose one colour without choosing its opposite in "temperature." So, if you pick a hot red, the opposite is a cool green. Or, if you pick a cool blue-green, you'll end up with a spicy red-orange on the other side.

One example of a site that consistently uses a warm/cool colour combination is [Ecolution](#), as seen in Figure 9.

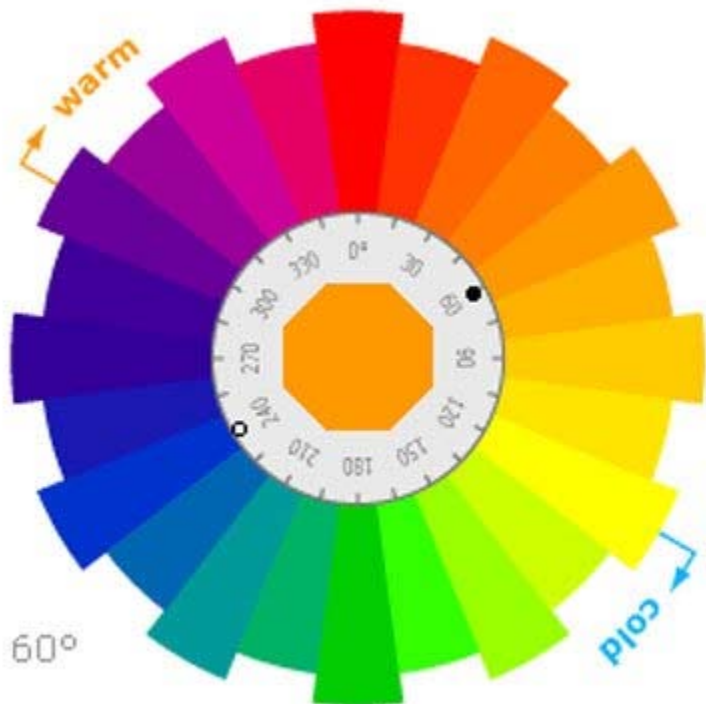




Figure 9: Ecolution—a good example of warm/cool colours.

Ecolution usually uses red as an accent colour on their home page in contrast to their green logo. They then blend the two contrasting colours with varying tints, tones and shades of those two colours. Even the “blacks” in an image can lean toward “warm” or “cool,” as do whites. Overall, the photograph is “warm,” which plays nicely with the stark pure green. Although they use the same colours as Greenpeace, the Ecolution site takes on less “glare” with the rich tones and shades in the photograph.

You never realized that colour theory was so easy, did you? Well, then...let’s complicate the issue a bit...

Triadic colour schemes

A triadic colour scheme (see Figure 10) is created when you pick one colour and then pick two other colours that lie equidistant from each other around the circle, like so:

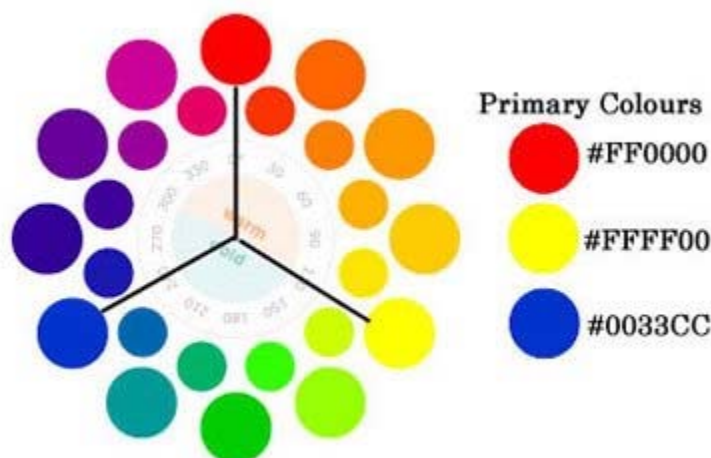
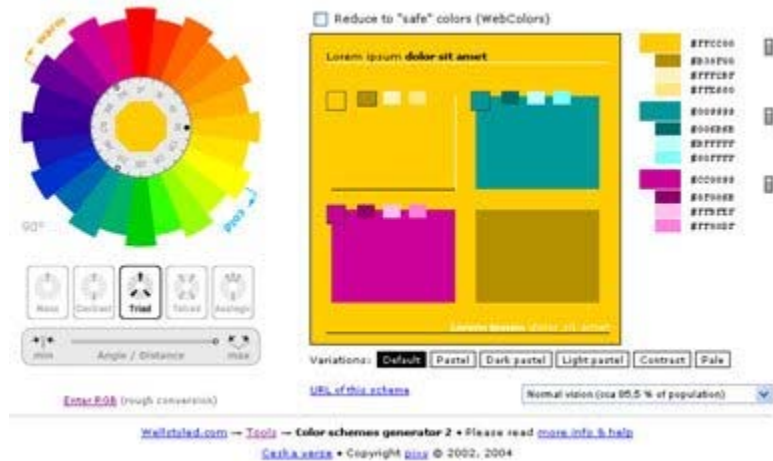


Figure 10: A triadic colour scheme.

I chose the primary colours for this scheme as I wanted to show how the colour schemes seem to contain a method to the madness. It’s no accident that the primary colours lie where they do on the colour wheel, as each colour contains an equal amount of secondary and tertiary colours between each primary. But, a primary colour triadic scheme can seem old-hat as it has been overused. Instead, you might try some other colour choices at the online colour generator, something like Figure 11:

Figure 11: An alternative triadic colour scheme.

The above triad scheme is built from orange-yellow, blue-green, and red-violet. I picked the orange-yellow first (notice the dark dot on the inner section of the colour wheel at left), and then chose the Triad selection located beneath the wheel. The generator automatically chose the triadic choices including all tints, tones and shades. The accompanying colours are marked on the colour wheel with the hollow dots, just as the complementary colour was noted in the monochromatic example.



Now, this is where a real colour wheel might come in handy, as the online results didn't quite match the results of a hand-held colour wheel. When I pushed the Angle/Distance tool under the colour wheel to "max," however, it seemed to match the colour wheel I held in my hand. The results shown above are those that matched the colour wheel the closest.

The triadic colour scheme also contains warm and cool colours, but one temperature will predominate. Usually, the temperature that will overshadow the other is the one that you chose on the front end. In this case, I initially chose the orange-yellow, which is a warm colour. The warm colours shown above will predominate as a result, with one of the other two colours lending the cool contrast.



Figure 12: Puzzle Pirates—A good triadic colour scheme.

[Puzzle Pirates](#), shown in Figure 12, uses a triadic scheme on their home page. They use the primary red-blue-yellow scheme, and this primary scheme is perfect for a kid's game site. Note that the blue is predominant and that the reds and yellows are used as accents and to lead the eye around the page.

Tetradic colour schemes

The more colours you choose, the more complicated the colour scheme. However, one trick is to find a tint, tone or shade and stick to those regions across the board rather than mix pure colours and their tints, tones and shades. This method works well with a

colour scheme such as the four-colour tetradic scheme. This scheme (see Figure 13) is just like the complementary scheme, only you use two complementaries that are equidistant.

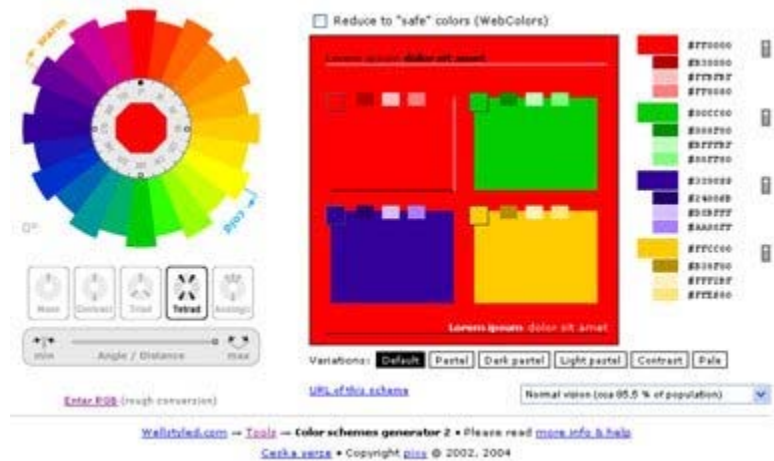


Figure 13: Tetradic colour schemes.

Figure 14 shows how a tetrad scheme works out online:

Figure 14: A tetradic colour scheme inside the online generator.

Note the black dot under the red in the colour wheel to the left. This was the first colour that I chose; I then clicked on the Tetrad button beneath the wheel. The four colours that showed up once again were a bit off from my hand-held colour wheel, but when I pushed the Angle/Distance tool under the colour wheel to "max," it seemed to match the colour wheel I held in my hand. The results shown above are those that matched the colour wheel the closest.



This colour scheme can become quite complicated, so what you might want to do at this point is to pick all four tints or tones or shades from the colours in the right column. You can make your choices by clicking the arrows at the far right. For instance, Figure 15 shows an example of a block filled with the tints of this colour scheme:

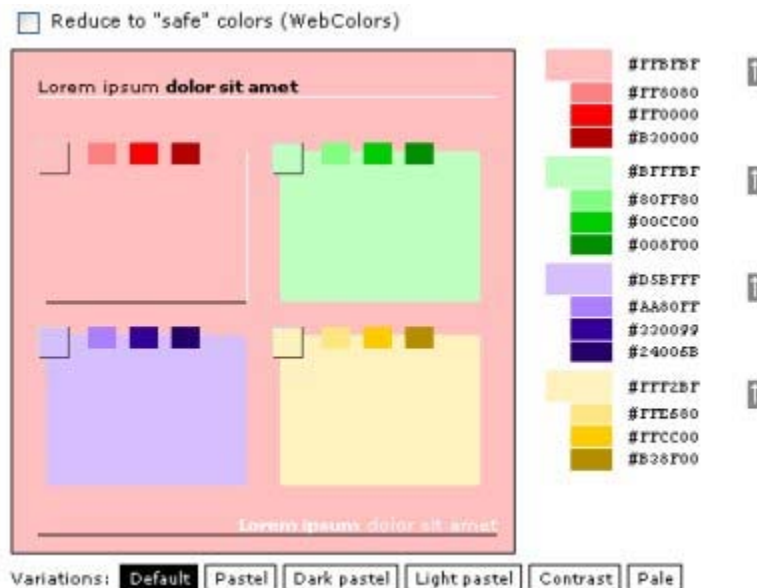


Figure 15: Tetradic tints.

And Figure 16 shows an example of the mid-range tones:

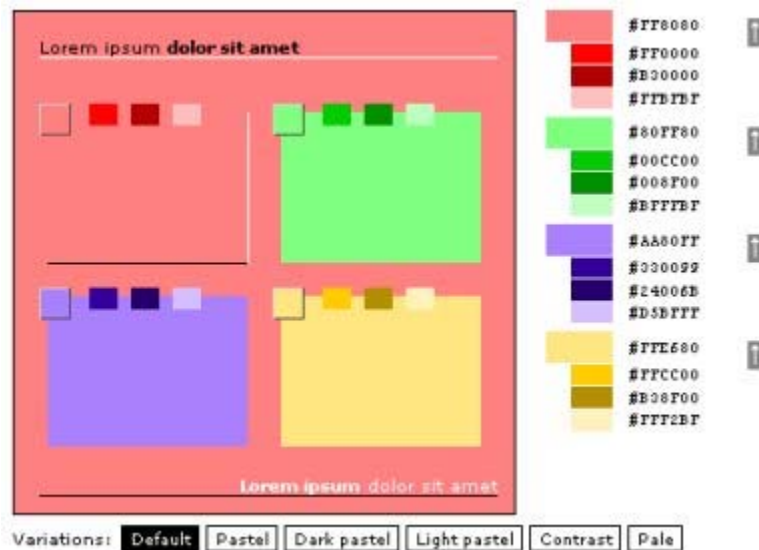


Figure 16: Tetradic mid-range tones.

If you look closely at the squares above, you'll see that the generator also provides you with four monochromatic colour schemes. These schemes are shown both in the column to the right as well as in each square within the larger square.



Figure 17: The Jane Goodall Institute site—a good tetradic colour scheme example.

The Jane Goodall Institute site (Figure 17) is one of the few sites that really carries the tetradic colour scheme well. Note the purple, the yellow tone, the red highlights in the photograph (the site holds more red further down on the page), and the greens. The purple doesn't match up exactly to the colour scheme generated by the online colour tool—it leans toward a red-violet—but it's close enough to use as an example of how you can use both a colour wheel and the online colour generator to produce ideas for your site.

Now, when you surf around the Web searching for colour and design ideas, keep your colour wheel close at hand to learn more about how designers use colour schemes on your favorite Web sites!

Summary

Although colour combinations may seem complicated, all colour schemes carry certain "rules." These guidelines make it easy to understand which colours work together to add interest and contrast to a Web site.

The reason colour wheels exist are for people to use them. Colour wheels and tools like the online Color Generator make colour-picking easy even for the inexperienced designer.

The four colour schemes covered in this article are monochromatic, complementary, triadic, and tetradic. Although other colour schemes exist, these four colour schemes are the easiest to understand and implement.

Exercise Questions

Note: the last two questions do not have "right" or "wrong" answers.

- Name the three primary colours, and explain why they're called primary colours.
- Name the three secondary colours and the primary colours that are used to make those secondary hues.
- Describe how a tint, a tone, and a shade are made.
- What is a monochromatic colour scheme?
- What is a complementary colour scheme?
- Describe "warm" and "cool" colours.
- What is a triadic colour scheme? Can you pick three colours that would fit this scheme?
- What is a tetradic colour scheme? Can you pick four colours that would fit this scheme?
- Which colour scheme seems easiest to use?
- Which colour scheme seems the most complicated?

About the author



Linda Goin carries a BFA in visual communications with a minor in business and marketing, and an MA in American History with a minor in the Reformation. While the latter degree doesn't seem to fit with the first educational experience, Linda has used her 25-year design expertise on site at archaeological digs and in the study of material culture.

Accolades for her work include fifteen first-place Colorado Press Association awards, numerous fine art and graphic design awards, and interviews about content development with The Wall St. Journal, Chicago Tribune, Psychology

Today, and L.A. Times. Linda is the author of several ebooks on Web design, accessibility, and—as a sideline—also writes personal finance articles and ghostwrites for a few SEO pros.

9: Building up a site wireframe

BY [LINDA](#) · 8 JUL, 2008

Introduction

Every web designer should know and understand a Web site's parameters before lifting a finger to start designing the site. In this article, you will learn the basics required to start designing business Web sites. While this information is useful if you want to build sites for others, it can also serve as a checklist article for sites you want to build for yourself. This is usually the stage that comes after [information architecture](#)—you should collect information on what the client wants on their site and how it should be structured, what kind of branding that company uses, and then use that information to build up a visual design mockup that you can ok with the client before you add graphics and colour schemes on to it. Specifically, I'll cover the following:

- Although colour and design are important, you need to understand what the client wants to accomplish with his or her Web site. This information will have a heavy bearing on the site's look and feel.
- You therefore need to manage a checklist of items to learn about the client's Web site before the design is attempted.
- You also need to know more about the company's previous marketing efforts, including branding. This information will have a bearing on the Web site design.
- Based upon all the information gathered from the client, you will create the visual design for the site so that the client can visualize the foundation for his additional graphics and content.

The article table of contents is as follows:

- [What you need to know](#)
- [The first steps](#)
 - [The imaginary example site](#)
 - [The logo](#)
 - [The layout](#)
 - [About advertising on a site](#)
 - [Checking the layout with validators and the client](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

What you need to know

Usually before a Web site design is decided upon, the individual or business should have a plan in place about what the Web site should accomplish. While colour and graphics are important, a plan should be in place for a budget, the intended market and projected goals as well as resources to accomplish these tasks. Is the site just going to give the user information, or is it intended to sell them products or services as well? Will this Web site expand, or is it intended to be a short-term effort to reach a niche market (such as a political campaign site or a site that intends to tap in on a current trend). Will the site include a blog, legal and information pages, photo gallery, e-mail contact form? What else does it need? How does this site compare to the competition?

Last, but not least among all these questions, is whether the company has a brand in place along with marketing guidelines. If not, then this job needs attention before a Web site design is attempted. The logo, branding of merchandise or services intended for a specific market, and a means to reach that market may be beyond your skills. If you haven't attempted this task before, you might pull in an expert in marketing to help set this business in the right direction. On the other hand, if a plan already is in place, then it's important to follow the company's directives so the Web site will fall in place with other marketing materials.

While much of this information may be decided before the proposed site reaches the designer, the answers to these questions can help you decide what type of site to design, the colours to use and the type of graphics to include. But, one thing can be determined up front in most cases—the site should be accessible and usable. Therefore, attention to code and to navigation is a priority in all cases. You can read more about accessibility later in this course (the main accessibility articles are still to be published), and some further points about usability issues from [Jakob Nielsen](#).

The point is to keep the site simple by using HTML and CSS for code and design, respectively. Avoid Flash unless it is appropriate for some elements of the site (much has been done to [make Flash more accessible](#) of late, and it is good for some tasks, such as video), and think about where JavaScript and other technical stuff is needed. This will make the site design easier for the designer and programmer to create (especially if the designer *is* the programmer) and it will be more compatible across all browsers.

The first steps

To help you along with these issues, I'll build a simple business site using a set of guidelines that I use for designing Web sites for myself and others. These checklists will include business aspects as well as design issues. For convenience, I'll use an imaginary business that is already developed, so it has used marketing materials in the past. Printed materials, including a logo and branding are already in place. If you're starting from scratch, the logo and branding plans will need to be developed first before you begin to develop the Web site.

As a Web designer, I'll want to know the following information about a business before I begin the site design. I want to make a list of everything that I want on that site design so I don't need to make radical changes later. This is not an imaginary situation, as the topics below need to be discussed with the business owners/decision makers to make sure that your vision of the project is in line with their vision.

1. Web site name: Does the name reflect the company and its online efforts? In this case, the Web site name is the company name, which is "Wiki Whatever." The company may want to develop a [tag line](#) as well if they don't already use one. The tag line will then be placed with the company name and logo together on the Web page.
2. Logo and branding: I want to collect any printed matter that was developed prior to this task, including logo, brochures, etc, so I can build a file that will hold information such as phone numbers and addresses. These items will also help me to understand the "tone", branding and style of this business better from their past efforts. If none of this has been developed previously, then I'll want to hire a logo design team to build a logo (I'm not a logo designer, so I farm that work out—and you can as well by building that price into your billing).
3. Web site domain name: In tandem with the Web site name, I want to know if the domain name is available. The domain name is the address that a Web site uses for identification, and that the user will type into a browser's address bar to reach that Web site. The domain name also is used as the link to the Web site from outside resources. The domain name can carry any number of [upper level domain registrations](#), such as ".com," ".org," and so on. While a designer usually isn't responsible for the domain name registration, it helps to know if the domain name has been chosen and registered. In some cases I've had to change a domain site name and some site content because the domain name was unavailable. This problem led to a higher charge for the client, which would have been unnecessary had the domain name been chosen first.
4. Competitor research: It helps to know what competitors' Web sites contain in terms of graphics and content so that the site you design will enter the market on an equal or better footing than the company's main competitors.
5. Information architecture: Does the site need a shopping cart or a blog? What plans for expansion does the site owner have in mind? What structure would be best to link the pages together? These items are important, as you will need to build them into the site design and its navigation. You need to know how the site will expand in the future—this will determine how you build the site as well.
6. Site content: Has the site content been developed? If so, you'll want to gain access to the content immediately to help determine the navigation, type design, and layout. Categorizing the content is the best way to develop navigation. The content can help determine the look and feel of the site as

well; therefore, if content hasn't been developed, it might be wise to delay the design. Be sure that the content is relevant and plan for updates, as site content is what keeps visitors returning to a site.

7. Research web hosts: While the client may have a particular Web host provider in mind, you may need something else entirely as not all hosts provide equal technology support. A Web host is the business that hosts Web sites, and some Web hosts provide access to databases, which you may need for a blog or for cataloging information or products through a shopping cart. Other hosts limit the number of visitors to a site, and this can create problems if the site becomes popular. For a large list of Web hosting providers and their capabilities, visit the Web Host Database ([WHDb](#)). Make sure the client has purchased space on this Web host before you begin your site design so that you know your design parameters.
8. Directed departure: Planning for directed departure means that you/your client gains control over how users will leave the site. Viewers will leave the Web site eventually, so why not plan for their departure through monetized ad placement or through link exchanges? Making plans for this direction now can add value to the site monetarily and/or through offering a service to your site users.
9. Deadlines: Determine now when the site will go "live". Usually an eight-week lead time is enough to finalize any small project such as this, as long as the clients have their content ready, they are amenable to colour and layout designs you offer as samples, and no difficult programming is required.

Once you have these basics out of the way, you can sit down, read the content, plan for navigation and decide how to best optimize the site for search engines. While you might not be in charge of SEO (Search Engine Optimization), you can work closely with an SEO expert to determine how best to use the [site's content and your code](#) to generate more traffic via keywords in content and in headings and subheadings.

Just as you wouldn't pick out carpets or a couch for a new home before the architect has created the blueprint, neither would you create a visual design for the site until you've planned the site's architecture. The navigation and plans for [SEO](#) in this initial stage will save time and headaches down the road. By the time you're ready to create a visual plan, you're already familiar with the site's direction and its contents, and this makes the job of working with colour and graphics that much easier.

The imaginary example site

This imaginary site is a business that provides Open Source code for wikis, and they come up with at least three new code ideas per week. Since the code is free to use and modify, the site owners want to monetize the site (generate money from it) through donations, ad placement, and through extra services offered by their programmers. The site name is "Wiki Whatever," and the domain name has been chosen. The content has been developed, and it contains code snippets that need to be cataloged, articles, and bios that feature the programmers involved with this project. The Web host provides [MySQL](#) database availability, and it is geared to accept the heaviest of surges in traffic without down time. Now it's time to pull together the items that will be used on the site:

1. Using the company's pre-existing logo, I want to prepare a digital copy to use throughout the client's Web site. I'll need a scanner to scan the image into a graphic program such as [Photoshop](#) or [Gimp](#). I'll size the logo for the site later, once I've determined the layout. I'll save the image at 72 dpi, which will allow for faster download time. I probably will use this logo for #4 below as well.
2. I'll use photos of the programmers on the staff page (or "About" page), so I'll need digital images for this project. They will either provide photos for scanning or send digital images. If they send digital images, I'll want an image with more detail than I'll eventually use, so a 300 dpi image is good, or a full-sized image that I can reduce later to my own specifications.
3. The client has decided to use a blog, since they already have sufficient content to keep this blog active for the next few months. Fortunately, the client has chosen a Web host provider that is familiar with blogs and has the necessary capabilities to handle databases and high traffic—including spikes in traffic. This host also provides several solutions for expansion, a great offer if the client wants to grow. If the host's up time is guaranteed, the client will be happier if he or she can stay with the same host throughout this growth phase down the road. This ability to stay with a host provider for years makes life much simpler.
4. Using FTP (there are several to choose from on the market, such as the Open Source products [Filezilla](#), or [fireftp](#) for Firefox, or a proprietary client such as [CuteFTP](#)), I'll upload a static page that announces the upcoming site. "Under Construction" is a terrible phrase to use, as visitors to the site

may not return if they don't know your "grand opening" date. Instead, use a page that states the name of the company, what they will offer, a date that the site will be active, and a contact (email is fine—if this is a bricks-and-mortar business, use a street address and phone number as well). Even better, utilize an email form that individuals can use to be notified when the site goes live. This last suggestion provides clients with potential consumers even before the site opens for business.

5. Using the content/structure information received from the client and the fact that they want advertising space designed into all pages, I'll design the architecture for the site and plan the navigation and textual links. I'll also use this copy to plan for keywords for the site's SEO.
6. Using the colours in the logo, I'll choose two or three colour schemes to present to the client for approval.
7. Then, I'll choose other photographs or illustrations from a stock photo place such as [iStock](#) or [Comstock](#). But, be sure to shop around, as some stock photo businesses offer sales and other deals that you may not be able to pass up. Using stock photography is not that expensive, and it saves headaches concerning [copyright issues](#). I'll also need any images that the company has produced—or that they will produce—to accompany any code, "how-to" articles and blog entries.

Note: These last two steps will be covered in the next article in the series; bear in mind that you want to get approval from the client for the visual mockup layout before you start putting colours and graphics all over it!

The logo

The logo is a vital part of any company's branding. While most businesses will not rush a logo as this piece of artwork will represent their business for many years, other clients will be less concerned with the image that represents a company. I can tell you from experience that a company that doesn't spend time and money on a professional logo usually will never spend that money—no matter how logical your arguments to the contrary.

The Wiki Whatevers company owners all attended Georgia Tech, so they used their Alma Mater's colours—gold and black—in the logo design. But, while the logo is simplistic, at least it may prove easy to work with in colour and layout. Figure 1 shows the logo:

Figure 1: The Wiki Whatevers logo.

The issue here is that I just scanned in a print logo, wanting to use the same logo online. The print colours, which are [CMYK](#) (Cyan, Magenta, Yellow and Key, or black), will not match the Web colours that consist of RGB (Red, Green, Blue). So, I'll need to try to do a little colour matching to get the Web colours to match the logo as closely as possible. There are four ways to accomplish this goal:



1. Contact the printer to ask what colours were used to print the Wiki Whatevers logo on previous printed matter. Usually, the printer will use [Pantone](#) colours, and Pantone provides tools that help designers match print colours to Web colours. This Pantone Colour Matching system is something that the printer may have on hand, so that printer can help match the print colours to the appropriate matching Web colour without spending money on Pantone tools.
2. Since the guys who own Wiki Whatevers used Georgia Tech's colours, I can go to the Georgia Tech site to see if I can match the colours from the Web. You can [use a graphic program to extract a colour from a Web site by making a screenshot](#) and bringing that image into a graphic program to use an eyedropper or some other tool to match the colours.
3. Eyeball the printed matter with the Web colours to try to match as closely as possible. In some cases, the colours may be vastly different. In other cases, the colours may seem too close to warrant a change.

4. Scan the print image in a scanning program that accepts CMYK and use Photoshop's Pantone Colour Swatches to match as closely as possible. This last solution works only if your scanner accepts CMYK and you own Photoshop software.

In my case, I was able to get that gold to match perfectly from the mascot image at the [Georgia Tech Athletic site](#). The gold is #eab200, and the black is, well, black (#000000). The background, which is a dark green-blue (#002123), was used in the drop shadow in the logo. So, what could have been problematic was made easy through a simple bumblebee mascot, as seen in Figure 2:



Figure 2: A portion of Georgia Tech's mascot used to match logo colours.

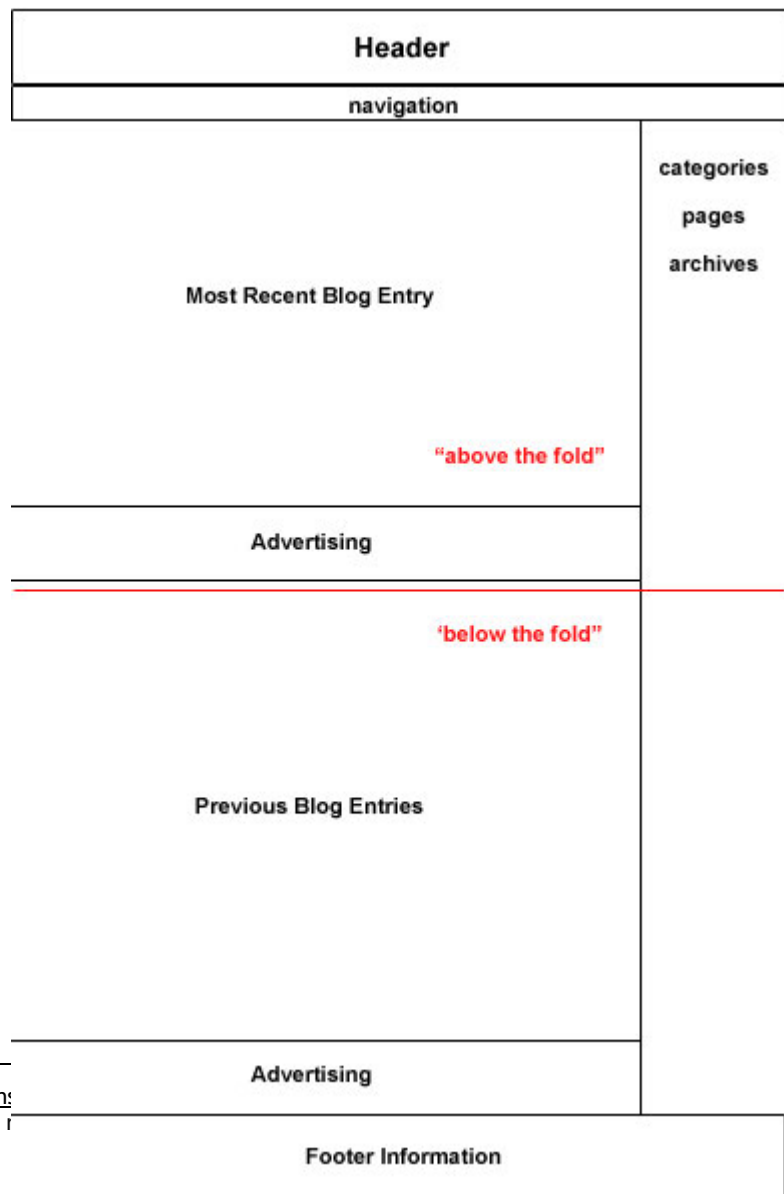
Note: Very seldom will you run across a business that hasn't used a digital image of their logo or brand online for items such as business cards and letterhead, if not for an actual Web presence. However, many of these businesses seem to accept the colours that the Web presents, rather than change the colours to match their printed matter. So, don't always rely on Web colours for a company's site, especially if those colours don't match the company print matter

such as brochures or letterhead. Instead, ask the company which colours they prefer—they may not have noticed that the colours were different in the first place.

The Layout

For the layout, and for simplicity's sake for this tutorial, I will demonstrate one layout. I chose a blog layout that promotes the most frequent changes to the body copy at top, easy access to navigation between the header and that recent body copy, and access to previous posts "below the fold" on the home page. The term "below the fold" originates from newspaper production. When a newspaper is on the stand, the reader will see only that copy placed "above the fold" in the newspaper (ie the physical fold in the paper). That copy—including images—is important, as it needs to entice the reader to purchase the paper.

The same "above the fold" theory applies to web site design. Anything that shows on a monitor when a visitor enters a site is "above the fold". Any copy that a viewer needs to scroll down to view is "below the fold". So, the trick is to keep a web site visitor's attention with the first images and copy that they view on any monitor, no matter the resolution (which is just one good reason to test your web sites on a number of different monitors/screen resolutions—an issue



addressed later in this article). A rough example for the initial layout for Wiki Whatever is shown in Figure 3:

Figure 3: Rough layout (wireframe) for the Wiki Whatever home page

This layout will remain the same throughout the site, but may change for the archived pages to list the article and blog entries without images. The reason for this consistency is so the viewer doesn't become confused. Once users "learn" how to use a site, they usually don't like to see changes from page to page. Here is what this specific design will contain:

1. **Header:** The header is small, as I didn't want to make the logo take up too much space on the page. While the logo is a minor feature, the colours within that logo will contribute to the overall colour scheme for the site. The header is at the top, a traditional placement, and the logo will link to the blog's home page. Linkage from the logo is convenient and many users have become accustomed to using the logo as a means to return to the home page on any given site
2. **Navigation:** Placed immediately below the header, the navigation is obvious and easy to use. This navigation also will be repeated in the footer. I repeat the navigation in the footer simply because I'm from the old school where navigation is repeated in simple text for those users who don't display images in their browser. Since I'm not sure at this point whether I'll use images for the navigation at top, I automatically include textual navigation somewhere else on the page as well—usually in the footer. This text helps those [blind readers](#) who use a screen reader to "read" a Web page. Whether you place that navigation text at the top or the bottom of the page is irrelevant except to the design, as blind readers can scan a page from bottom to top and vice versa as quickly as a sighted person. With that said, it is up to the designer and his/her client whether or not to repeat the navigation anywhere on a given page. If you use images for navigation and do not repeat the navigation links in text, then be sure to include descriptive `alt` attributes for those navigation images. This way, users who use a screen reader or who turn off images will still know what those images are for. See the relevant section of article 17 for more about [correct use of `alt` attributes](#).
3. **Most recent blog entry:** The most recent blog entry deserves highlighting, and the ability to make this entry a major focus on the page "above the fold" is advantageous to both the client and his readers. As soon as the viewer surfs to this site, this is the copy that he or she will see. This obvious placement, however, means that the client will need to continuously update the site on a consistent basis or risk losing return visitors—people are unlikely to keep returning to a blog if there is no new content being published on it.
4. **Previous blog entries:** This is where previous blog entries will reside—about three to five entries should be enough to let viewers know, at a glance, exactly what to expect from this site on an ongoing basis. Images might be nice here but they are not necessary as this area is located below the fold, so it is not as important for catching the viewer's eye. The decision to use images may be based on whether download time is an issue, or if the previous article actually needs an image to entice the viewer to click through to the full article/blog entry.
5. **Right column:** This is where viewers can gain access to blog entries listed by category, archives, and other types of site content. Examples of other pages include an "about the company" page, a site index and contact information. It's important to decide how you want to list these items in a side column, as the blog will build upon the categories you create, the pages you build and the archival material. As you grow the site, these lists will become longer, possibly to the point where—in this case—the categories may be the only list that a viewer will see "above the fold". The clients may decide that the "pages" are more important than the categories, and the list shown above may be altered to place pages above categories. As a side note, this list does not include everything that can be included in a side bar or side column. Some clients may feel that two columns would be better, making the blog a three-column, rather than two-column blog as shown above.
6. **Footer information:** Footer information is vital and important, as it provides viewers with background information about company and its Web site at a glance instead of having to dig for it. Company name, possibly a repeat of the logo, address, email address, links (to contact form, privacy notices, disclaimers, legal information), and news summaries are all good candidates for being

included in the page footer. As mentioned before, you can also repeat the navigation as a text only version.

7. **Advertising:** In this layout, the advertising is positioned following the recent blog entry and the previous blog entries in a horizontal banner. This provides the client with the flexibility of choosing ad text or banner art for their advertising needs. This type of layout for advertising places just one ad “above the fold” and another ad “below the fold”. This amount of advertising is plenty enough for most sites. Additionally, it relegates advertising to a secondary position, below the body copy for the site’s contents.

This layout allows the viewer to quickly move from body copy to navigation without scrolling, and it also allows users to view other topics that the site may cover with further links into the site’s categories at the very least. Even if the viewer never scrolls down past the red “fold” line, the layout provides all the major elements that a viewer might need, all placed “above the fold.”

About Advertising on a Site

It is to the client’s advantage, and a service to the reader if the advertising on a site is content-relevant. In other words, if the content on the site is about flowers, then ads for that site might include landscaping services, catering (to compliment floral arrangements), etc. So, for a site that provides open source materials, you might seek advertisers that are relevant to open source content. [Google AdSense](#), as one source, would help in this regard, as the ads are content-relative. The use of this type of advertising is a great idea until traffic grows enough to entice other advertisers to your site. However, always think about the SEO implications before you accept ads, as some advertising may adversely affect the client’s standing in search pages. Some good SEO resources are as follows:

- [Intelligent site structure for better SEO!](#), by Joost de Valk.
- [Semantic HTML and search engine optimization](#), by Joost de Valk.
- [How Affiliate Programs Can Affect Search Rankings](#), by Fredrick Marckini.
- [New Report Explores how PPC Rank Affects Traffic](#), by Jennifer Laycock.

Note: You may not be responsible for site advertising as a designer unless you’re designing a site for yourself, but if you plan to work with an advertising or design agency in the future, you may want some input into the advertising at these design firms. The more you know about what makes a site successful, the more success you may encounter in your design career. When possible, learn as much as you can about marketing (for yourself and for your clients) and search engine optimization tactics.

Checking Layout with validation and the client

Before this layout is implemented using code, I want to confirm the layout(s) (or wireframes) with the client. One tactic I use to convince the client that any one layout is better than the other is to remind them that coding additional layouts does cost money. This helps the client pick one layout, with the idea that the code can be tweaked later to make some structural changes.

The next step is to code the layout and then validate that code. I use the W3C [Markup Validation Service](#) and the W3C [CSS Validation Service](#) to confirm that the HTML and CSS used to build the layout contains no errors. You can upload files directly from your computer to these validation services, so you don’t need to upload them to the client’s site to test them. This test allows the designer and/or the programmer to find any errors on the front end that can be resolved before the site becomes saturated with code from images, advertising and other items added to the pages.

Accessibility is another big concern—making sure the site is usable by people with disabilities such as blindness or motor deficiencies. This isn’t as easy a validation process as validating your CSS and HTML. There are checkers available, such as [TAWDIS](#), but ideally you should test with real users and do a qualitative analysis of your sites for accessibility—a mechanized checker can’t conclusively say whether a site is accessible or not, although it can give some indication as to what is right and wrong. They sometimes make mistakes as well. There will be much more information about accessibility published in the next part of the course, so watch this space!

You should also test the layout across the different available browsers so that you know your web site can be viewed by the maximum number of web users possible. You can do this by having Mac, Windows, Linux and Mobile systems available, all with various browsers installed, or use emulators such as [VMWare Fusion](#) to emulate different systems on one computer, but this is rather fiddly and longwinded. Another option is to use browser-capturing services such as [BrowserCam](#), as this service is fast, convenient, covers a number of different browser possibilities (including much older browsers). They offer a 24-hour free trial so you can see if this service is right for you, and after the free offer, the charge to use this service is worth it, especially if you design a large number of sites and test on a large number of different browsers.

Finally, it's a good idea to check in with the client to let them know that the code has been generated for the layout and that it has been validated; you should also let them know how many changes were needed to the wireframe to get it to work across browsers. Only after the code is generated, validation finalized, and go-ahead from the client received should you begin to add colours, images, and any other code such as for advertising. Although this work may seem tedious, it's best to confirm all the validation and get client approval before adding the icing on the cake. Otherwise, you may find yourself working harder than you want as you find code problems and browser incompatibilities after the fact. Additionally, any artwork or body copy can prove a distraction for the client when they are trying to review the actual architecture of the site.

Once you've completed this process, you can then begin to work on the site's text, images, and colours. How do you begin to do this? Find out in the next article!

Summary

The web designer often wears many hats, because Web site design is based upon many factors. Will that site grow over time, or will it remain static? Can the Web host provide consistent quality service and room to grow, or will that client need to move from Web host to Web host with new additions to the site? And, if the designer cannot sufficiently perform all design issues, does that designer have a network of people on hand to help?

So, beyond colour and graphics, a foundation needs to be laid to build that Web site upon. The business of building a site affects the design, and any issues that might become problems down the road can be ironed out in the planning phases. This ability to resolve issues before they arise makes for a professional designer.

Once the foundation has been laid and the architecture and wireframe of the site has been developed, the designer then can begin to work with colour schemes to develop the full Web site for client approval.

Further reading

Here are some other sites that offer checklists:

- [Dive In Designs checklist](#)
- [Net Mechanic's Web Usability Checklist](#)
- [Max Design checklist](#)
- [Usability First's Checklist](#)
- [David Skyrme and Associates' Checklist](#)
- [SCORE's Web site design checklist](#)

Exercise questions

- What items should you have on hand before you begin to develop a Web page design?
- Why should you list all the items that you plan to use on a Web page?
- Why is it important to research Web host providers?
- A designer can wear many hats, but what would you do if your client asked you to design a logo and you didn't know the first thing about logo design?

- Name two good reasons to research a company's competitors' Web sites.
- What is CMYK and what do those letters mean?
- Name at least two ways to convert CMYK to a matching RGB colour.
- Name one reason why a designer should use text for navigation on at least one area in a Web page layout.
- Why would a designer keep a consistent layout for a Web site throughout that site?
- Name one reason why a site's code should be validated in the early stages of design.

About the author



Linda Goin carries a BFA in visual communications with a minor in business and marketing, and an MA in American History with a minor in the Reformation. While the latter degree doesn't seem to fit with the first educational experience, Linda has used her 25-year design expertise on site at archaeological digs and in the study of material culture.

Accolades for her work include fifteen first-place Colorado Press Association awards, numerous fine art and graphic design awards, and interviews about content development with The Wall St. Journal, Chicago Tribune, Psychology

Today, and LA Times. Linda is the author of several ebooks on Web design, accessibility, and—as a sideline—also writes personal finance articles and ghostwrites for a few SEO pros.

10: Colour schemes and design mockups

BY LINDA · 8 JUL, 2008

Introduction

After a web designer presents a site's architecture, or wireframe, to a client for approval, the next step is to determine the look and feel of the site through colour and graphics. In this article, I'll demonstrate how I keep this process as simple as possible, both for myself and for the client. I believe in the KISS philosophy—Keep It Simple, Sweetie. There are two reasons for the use of this tactic: First, life is complicated enough without adding more curves; secondly, a simple plan also helps to keep a site more accessible and usable for all concerned. In this article, you will:

- Determine the typeface for headings, subheads and body copy, along with any other typographical issues.
- Plan different colour combinations for the web site.
- Build a mock-up for the client to decide on the colour and graphics.
- Consider how to test the web site before it goes "live".

The article table of contents is as follows:

- [First step: considering typography](#)
 - [About typefaces or fonts](#)
 - [About readability and legibility](#)
- [Second step: add typography](#)
 - [Attention to alignment](#)
- [Third step: colour](#)
- [Fourth step: testing](#)
- [Summary](#)
- [Exercise questions](#)

First step: considering typography

You will learn some more about the technical aspects of typography in the [typography article after this article](#); now I will look at a few aspects of it that you will find useful at this stage.

[Fonts](#) also are called "typefaces" and they're used to display text, numbers, characters and other symbols. Also known as "glyphs," these symbols, letters and numbers are categorised by family (all related), style (italic, normal, oblique, etc.), variant (normal or small caps), weight (boldness), stretch (condensing or expanding type by height or width) and size (by points or by pixel height or width). Typography is the arrangement and appearance of text, so typography concerns the look of the glyphs and how they are placed on the page (columns, paragraphs, alignment and more). The most effective way to control how typography looks on a web page is through using Cascading Style Sheets ([CSS](#)).

One of the first steps in finalising a web design is to decide upon the [fonts](#) you will use throughout a web site. Many studies have shown that a variety of typefaces on a web site may confuse the reader. On the other hand, a site that uses only one font throughout seems bland.

My advice is to use one typeface for the headings and subheads and another typeface for the body copy, especially if you plan to add advertising to a site. Limiting the number of fonts adds continuity to the site while still allowing the reader to differentiate between headings and body copy when they're scanning the web page. Advertisers will add the variety; you can never know in advance which typeface or variety of typefaces an advertiser might use in a banner or text ad. Personally, I usually use Verdana for body copy and Times Roman or Georgia for headings. The Times Roman and Georgia typefaces are serifs, and Verdana is a sans-serif. As you'll see shortly, I decided to stay with Verdana for body copy in the example site, but since the logo was built upon Arial Black, I also used that sans-serif typeface for all the headings.

Sometimes you need to break your own rules, and this layout provides a prime example for that contingency. But first, allow me to explain the difference between typefaces, and why I keep my choices so simple.

About typefaces or fonts

There are four main types of fonts and they are as follows:

1. **Serif:** Any typeface that contains a finished stroke, flared or tapering ends, or have actual serified endings (including slab serifs). Throughout history, the serif font has been chosen for print body copy, as it's easy to read on a printed page. But, the Web is different than print, and [some studies](#) on [font readability](#) show that some sans-serif fonts are easier to read, like the one shown in Figure 2, when they're used as body copy on a web page.

Times New Roman, regular, 18 point

Figure 1: A sample of serif type, Times New Roman, regular (not bold or italic), at 18 pt.

2. **sans-serif:** Any typeface that has stroke endings that are plain without any flaring, cross stroke, or other ornamentation. While [some authors](#) argue that the studies on font readability are flawed, I've noticed that those sites use sans-serif type in their body copy. Even articles that argue that [serif typefaces are more legible](#) use sans-serif type in their web page copy. So, I tend to follow the crowd here—I use sans-serif like the one shown in Figure 2, because it has become a traditional body copy typeface for the Web.

Verdana, regular, 18 point

Figure 2: A sample of sans-serif type, Verdana, regular at 18 pt

3. **Script or Cursive:** These fonts usually look more like handwritten pen or brush writing than printed letterwork, like the example shown in Figure 3 below. These fonts would include those that appear handwritten, even when they aren't cursive. One reason not to use this type of font on a web page, especially in body copy, is that it is difficult to read in large chunks (think about how difficult it was to read that hand-written letter from your aunt Margaret, or the manuscript from the twelfth century that you once viewed in the museum). Additionally, not all browsers display the same fonts, so if you decide to use a script or cursive font, it may choose to show itself as a serif font in someone else's browser.

Staccato222BT, regular, 18 point

Figure 3: A sample of the script font, Staccato, at 18 pt

4. **Specialty, including monospace:** The sole criterion of a monospace font is that all glyphs have the same fixed width, similar to what a typewritten page might look. Other fonts might have a fantasy appeal, like the one shown in Figure 4 below, and these fonts are used solely for decorative purposes. Monospace fonts have their place on a web site, especially when [displaying programming code](#). This typeface often is used for that purpose, as a monospace font clearly shows each letter and symbol used in code.

JOKEWOOD, REGULAR, 18 POINT

Figure 4: A sample of a specialty script, Jokewood, at 18 pt

A look at the visuals above will show that not all type is created equal, even if it's created at the same point size. The point size is the height of a letter, and some fonts will be larger at 18 points than others.

Other variances exist as well, such as the spaces between letters and words, or the fact that some typefaces, such as Jokewood, lack lower case letters. But, you can see where Jokewood and Staccato, the script, would be difficult to read as body copy. Those scripts might find a place in small areas as headings or in advertisements.

One point to consider is that these fonts may not show up the same across all browsers because different browsers basically remain incompatible. You may have picked up on my comment above that “not all browsers display the same fonts.” The reason for this problem is that not all operating systems support the same fonts. Or, they may support the same fonts but the variant, weight, and other factors may show up differently from one browser to another. For this reason, you may choose to use a [generic font](#), or simply a “serif” or “sans-serif” to display your web page’s typography. Or, you can use both the generic name and the name of the font you chose and hope for the best, because users also have the ability to change your font or the way it looks in some cases.

The only way you can use a specific typeface in a specific style, variant, weight or stretch across all browsers is to create a graphic with that font in a graphic program. This is, however, not recommended for many reasons, including the text being inaccessible (a screen reader cannot read the text hidden inside a graphic), harder to resize (not all browsers do full page resizing), and harder to maintain (you’ve got to recreate the graphic every time you want to change the text). Trust me—don’t go there.

It is for this reason that, as a web designer, you must learn to let go of the fact that Web is—for all intents and purposes—basically a malleable format. You have control over only so many things in a web site design. Type is one item that you have some control over, but only if you keep it as simple as possible. And, that is the reason why I’ve stuck with Verdana for body copy and Times Roman or Georgia for headings for all these years.

With that said, type designers and programmers continuously try to find ways to make fonts [more legible and beautiful](#). So, take what I say with a grain of salt and try something new if you think it will work. You’ll find out soon enough if it doesn’t pan out when you test that web page across a number of browsers yourself (a task I’ll explain later in this article).

About readability and legibility

When you show your web page design to a client, he or she usually won’t know the difference between a serif or a sans-serif typeface. All that client will know is whether or not he or she can read the content on that page. So, in the end, legibility is what matters. To that end, you need to make sure that:

1. Your type is large enough to read in a variety of browser resolutions. Although users have the means to change a font size in some browsers (such as Opera), you can try to make sure that your fonts are adjustable to different [browser resolutions](#). One way to accomplish this feat is to size your fonts by [percentages](#) or ems rather than by pixel height (you use CSS to do this).
2. You provide enough contrast between background and body copy. White or light type on a black background for large chunks of body copy can [strain some eyes](#), however - if you need to go that route, try to provide an alternative stylesheet so users can read dark copy on a light background instead.
3. The headings are, indeed, different than the body copy. The ability to break up large chunks of copy with headings and subheadings, or with lists (like this one) makes it easier for users to scan a page to find what’s important to them. Breaking up a page with images is nice, too, but make sure that those images are relevant to the copy. Otherwise, you’re just wasting bandwidth.
4. You avoid stretching body copy across the entire width of a screen with a fluid layout. Try reading a paragraph that stretches the horizontal space provided by a large monitor—it will wear you out after a while because you’re using your eyes *and* your head as you scan the text from one side of the screen to the other side. Take a look at [this web page on readability](#), which is the best I’ve seen to date to illustrate a typical readable width for body copy (illustration shown below in Figure 5 as well). This page also explains in depth how people read a page, no matter if it’s web or print. The image shown below was captured on a twenty-four inch screen at a 1920 x 1200 resolution. Compare this image to the one you see on your screen when you click on that previous link. Then, note the

resolution of your screen to see the differences. Sometimes a fixed layout is a good thing, as it will define the parameters of the body copy area for the benefit of your readers. Don't worry about all the space around that body copy area (like that shown below). Provide a nice background that doesn't distract from the design or the copy, and you'll provide a service to those who own large screens.

40-60 characters is generally regarded as a good line length for body text, and this varies depending on factors such as font size and target audience.



Figure 5: A sample of an appropriate width for body copy, displayed on a large screen.

Last, but not least, be sure to check your body copy and headlines for typographical and spelling errors. I found at least one typographical error and a few spelling errors in approximately half the links I provided thus far for this article. While it's human to err, a company's or an individual's credibility can be marred by poor spelling and with words that lack

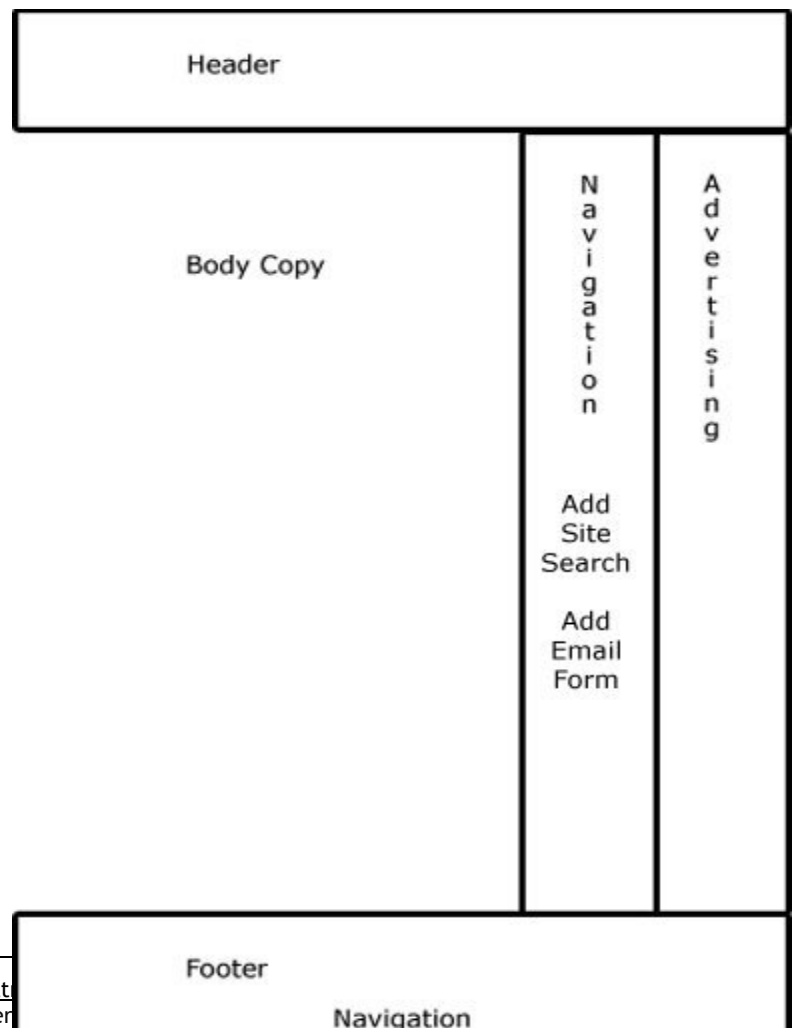
spaces or other issues that can easily be resolved.

Second step: add typography

Once you've chosen the fonts you'll use for the site, you'll need to layout the headings, subheads and body copy in the site layout. In the previous article, I introduced an imaginary company, "Wiki Whatevers", a business that wanted to share their code snippets on a web site as open source material. I designed a site architecture for approval, and I am happy to report that they loved the ideas that I presented! Although this wireframe seems stark, it avoids any placement of graphics or images that may give the client a preconceived notion of where things "might" go, including the company logo. The image of that wireframe is shown below in Figure 6:

Figure 6: Wireframe for Wiki Whatevers

Now, for my own piece of mind, I want to add the logo and some of the copy that company provided to determine if it will flow correctly within that layout. The



other reason I want to add the logo and the copy at this point is so that typography can lend a hand in the colour choices I'm about to make. Body copy, headlines, and other typography on a web page adds its own "colour" to a site. Just compare the image below in Figure 7 to the one directly above in Figure 6 to see the difference:

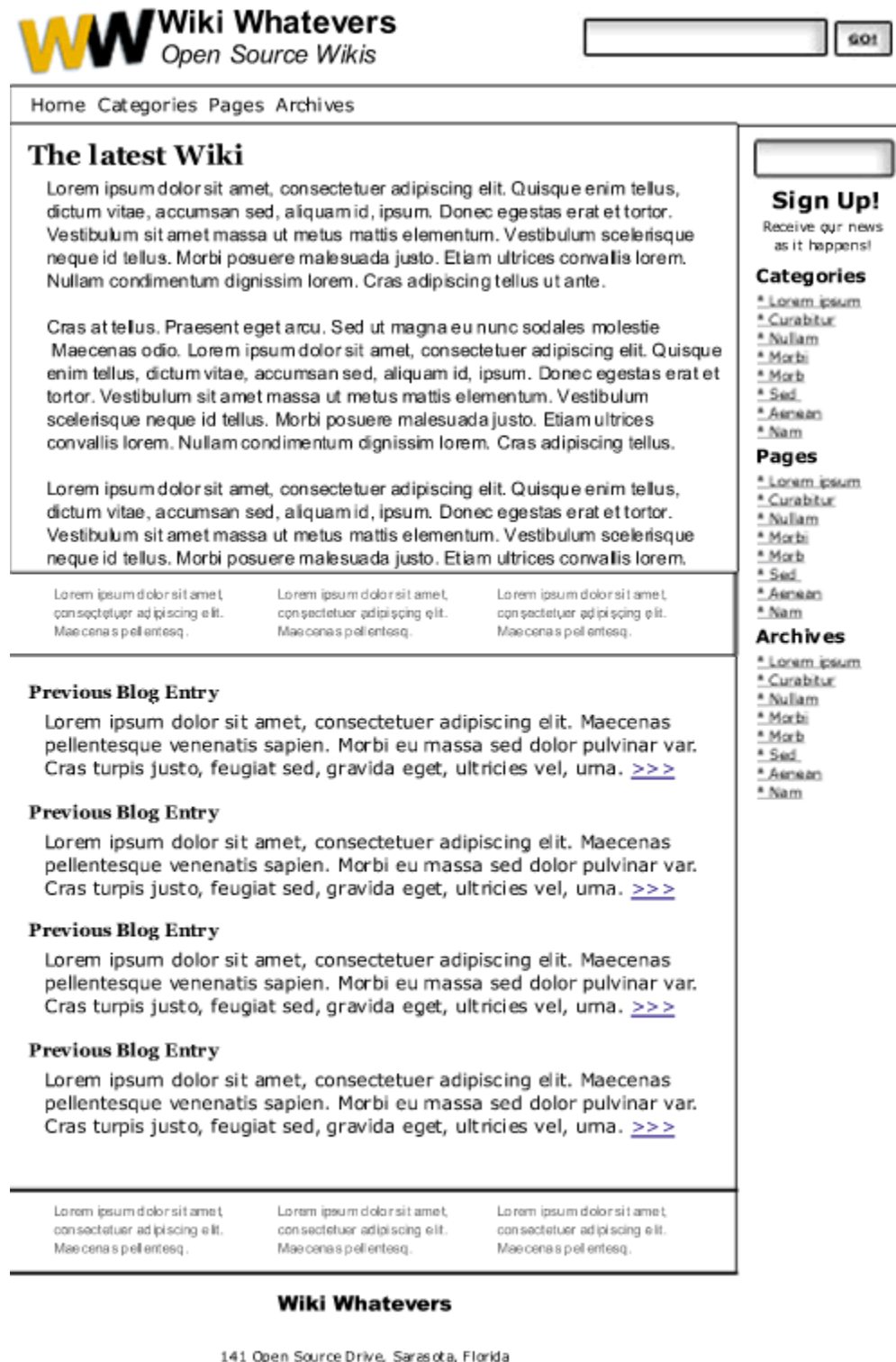


Figure 7: Body copy and logo added to wireframe

The copy above adds “colour” to the site, although the only colour has been added by the logo. The copy has added “value”, not just in content, but in tints, tones and shades of black and white and grey. I’ve added only those items that are essential to the site—the logo, the site name (which also is the company name), a tag line (Open Source Wikis), the links and sign-up for a newsletter or feed in the middle column, and the footer information about the company as well as text links and ad copy. The search box proved too large to use just one column, so I placed it into the header.

While I say I used body copy, in reality I used a “filler” provided by the [Lorem Ipsum generator](#). This “fake” body copy can stand in for real copy if you want to design a site and the copy remains unavailable to you.

Attention to alignment

At this point, I want to show how I made the decision to place the copy where it is as far as alignment. Alignment means how copy is positioned in a certain area. You can align it to the left, and leave “ragged” edges to the right, which is a traditional layout. Or, you can center the copy, or make it justified (aligned both right and left equally), or align it so the right edges are straight and the left edges are ragged.

I chose to use the traditional left alignment, where the type lines up in a straight line to the left. But, you might notice that the body copy is indented further to the right than the subheads that go with that copy. The logo was the reason why I chose this alignment. Here’s a close-up illustration that explains why I made that decision in Figure 8:

Figure 8: Illustration of alignment between logo and body copy

Since the logo isn’t an award-winning design, I tried to keep the size small. Plus, the weight provided by the Arial Black typeface makes the logo appear larger and more out-of-proportion than any other feature on the page. Although I reduced it greatly, I wanted the logo to be large enough to fit the company name and their tagline, “Open Source Wikis”, into the space provided by the height of the logo. You can see from the red lines I placed in the illustration above that the business name is aligned at the top of the slanted logo, and the tagline is located at the bottom of the black “W” in the logo. (Also look back at Figure 7 - looking down from the first “W” in the logo, you’ll see that the heading for “The latest Wiki” is aligned with that letter’s left bottom point. Since the logo is slanted, the lowest point helps to point to the heading below the navigation. In fact, it makes the navigation appear secondary, since the heading is as bold as the business name.)



At this point, I noticed that the Georgia typeface that I chose for the blog entry title seems a little too busy for that area. This is when I decided to change the headings to Arial Black, a sans-serif type that is slightly different than the Verdana I used in the body copy, yet not so different as to create chaos.

This ability to align various elements within a page design is made easier with CSS. Although some browsers may try to thwart your efforts at alignment, in many cases using exact points within a layout to apply your elements will work. This is why you also can align a search box with the bottom of a tagline, or why you

can align the right side of that search box with the right side of a form located below—just as I did in the sample shown above.

I could have aligned the body copy with the logo's top left corner and with the headings as well, but the indentation shown above allows the reader to quickly scan headings and read the copy that's important to the reader. With that said, each web page is different, and another logo design might call for a totally different alignment. The point is to align all the important elements on the page so they flow smoothly. The reader won't notice that you've gone through this trouble (neither will the client, in most cases). But, the minute you don't take the time to align important objects on a page, someone might make the comment that "something doesn't feel right".

Moving body copy in by a small amount also creates what is known as "white space", or a shape (sometimes called a gutter) around the copy and any other elements that allows the viewer to easily perceive one piece of copy apart from another. The space on the left of the body copy should usually be set similar to the space to the right of the body copy (or any images), so that you can create a balance within a given space. This white space gives the site some "breathing room", so that it doesn't appear too crowded.

Take a look at web pages the next time you're surfing around on the Internet. Note the pages that seem well-designed to you, or those pages that just "feel right". Note the alignment of body copy, headings, logos and other elements on the page. Are they aligned? Do they seem to flow from one element to another? Perhaps the designer took the time to attend to small details like this to help make the page design less important than the content—which, for all intents and purposes—is the goal of good design.

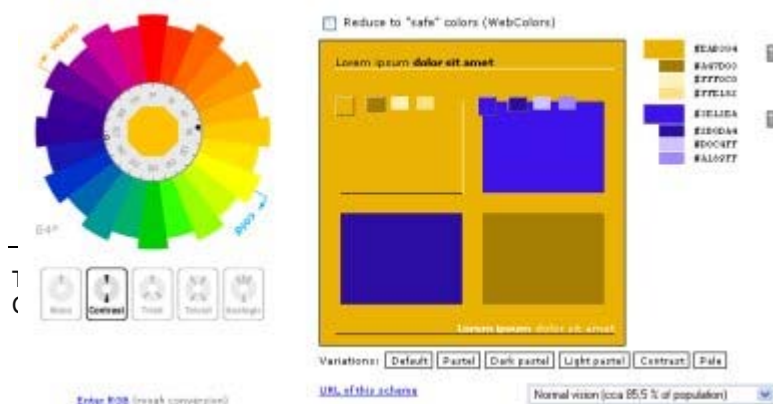
Granted, a design may encounter some issues in different browsers. The text may not line up just right in Safari, but it may appear perfectly aligned in Opera or Firefox. That's a road to cross shortly, but only after the colours for this site are chosen.

Some of you may wonder why I didn't choose Arial Black for the subheads as well as for the name of the company. I chose to remain with my faithful choice of Times New Roman as this serif face adds contrast to the site, a design principle that adds interest to a page. The overall use of Arial Black for all the headings would make this site appear bland.

Third step: colour

When I prepare a web site mock up for a client, I usually try to take it as far as possible after the initial wireframe has been developed before I show the samples to the client again. When possible, I prefer to use code rather than an image for the wireframe. This way I can drop in elements such as the logo, body copy, and even mocked-up ads to show the client how everything will appear on the final page. With such a complete layout, the client has no delusions about what the page will look like with all elements in place. Then, the client can make decisions about what can be added or removed. Additionally, when I can show a client a web page on a computer just as it would appear on the web, the client can visualise how that page would look if he visited it in real-time.

Colour is part and parcel of that "everything in its place" mentality. The reason for this is that different colour schemes can change the mood of a site entirely, even with all elements in place. Further, I prefer to keep the colour samples to a minimum, because too many samples can become confusing. In this case, the client had a limited budget, so I persuaded them to limit the choices to one colour scheme as a sample to work with.



When I introduced the [Color Scheme Generator II in Article 8](#), I didn't mention that you can input a hex colour in this tool to generate a colour scheme from a specific colour. Directly beneath the colour wheel, you'll see a link for "Enter RGB". In my case, the gold in

al - Share Alike 2.5 license.

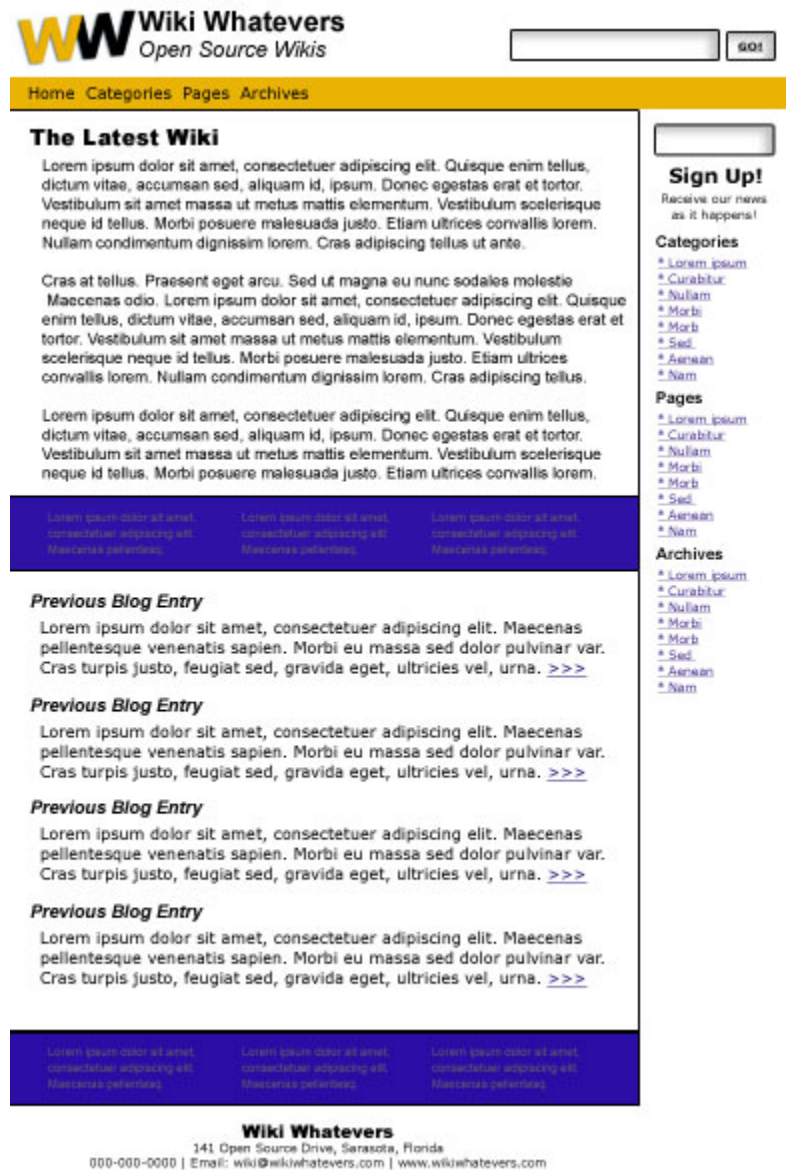
the logo was the strongest colour, so I entered that hex (#eab304) to learn more about my choices. The resulting monochromatic scheme was somewhat boring, but the contrasting colour scheme held promise. That scheme held a blue-violet that I could work with, as the shadow behind the logo was tinted blue as well, as shown in Figure 9:

Figure 9: Contrasting colour scheme based upon #eab304

With the colours shown here, I decided to use the main gold logo colour as the background for the top navigation. I used the darker blue (almost a blue-violet) #2b0da4 for links (which I also underlined), and I used a lighter opacity of that same colour blue for the advertising background. You can see how these colour additions alter the look of the layout in Figure 10:

Figure 10: Layout in contrasting colours

You can see in the image above that the colours are far too dark and “heavy” for the site. So, I took the opacity in the navigation bar down to 75% and the opacity of the colour in the advertising copy bars all the way down to 20%. You can see the immediate difference below in Figure 11:



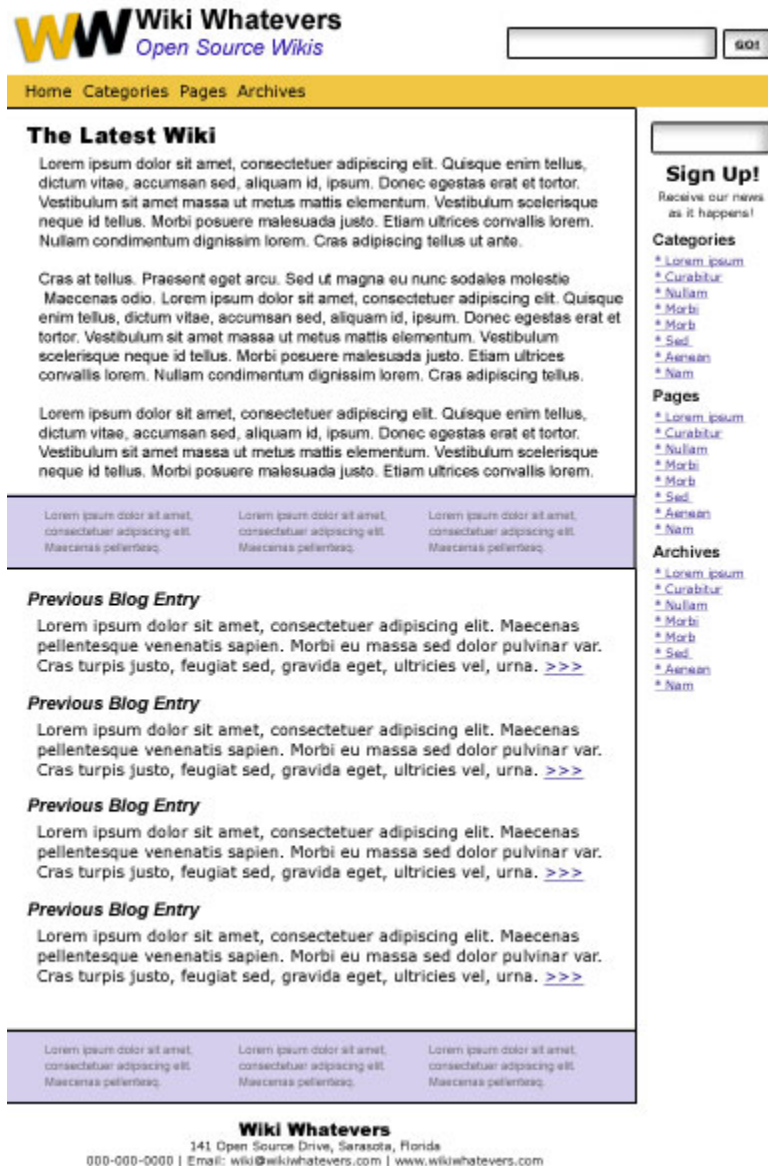


Figure 11: Layout with reduced opacity

The reduction of colour in the navigation bar brings it more into alignment with the colour in the logo. The reduction of opacity in colour for the advertising background brings it in line more with the link colours. Since the ads consist of links, this colour reduction and matching to the site's links is appropriate. The fact that the colour behind the ads has been added is a good thing—if you use an advertising service like Google AdSense, you will learn that Google prefers that you make the ads stand out from the rest of the body copy—the colour addition helps to meet that criteria. I also used the #2B0DA4 dark contrasting colour for the tagline, and this addition helps to bring the blue contrasting colour around full circle on the page.

While this layout may look like it was easy to pull together, I spent some time using the colours from the contrasting colour scheme for backgrounds, to colour the headings and to change the ad layout several times. With each change, the colours seemed to overwhelm the simple layout, so I eliminated them and stayed with the black for all typography except the tagline. While I could use a “visited” colour link, I believe my best choice is to stay with a simple two-colour base that will accept colour images readily without creating a colour nightmare.

On the other hand, you can see how building a wireframe up front can simplify your work as well—once you have a “map” or architecture in place, the addition of colour is more or less like colouring “within the lines”. If you stick to your layout, you can use that layout to dictate your choices. In addition, if you keep it simple the design can be more elegant in the long run as well as more usable and accessible.

There is one last good reason to keep this layout simple—the inside pages will carry code snippets, and I’ll use a monospace font for that code to be in-keeping with best practices. This is another reason why I chose to use two different sans-serif typefaces that are similar in style. The application of a monospace font inside this site will add enough typeface variety, along with any advertising variants. Make sure you use the heading elements (`h1`, `h2`, `h3`, etc.) for headings and subheadings rather than making them bold (`strong`) or italicised (`em`). Using heading elements will make your site more accessible. You can alter how those headings appear in your style sheet (CSS).

A few things to note about the page above:

- I kept the company name at the top of the page in black, as that black helps to carry the black in the logo across the top of the page.
- I used a centered alignment in the top of the middle column to draw attention to the registration feature. Since the form for registering for a newsletter fills the width of that column, the centered copy is balanced by that alignment, and it makes the copy seem to “belong” to the shape created by that form.
- The centered alignment of the address at the bottom of the page in the footer may seem off-center, but I wanted that address to become part of the area that carries the body copy for the site. As the site grows, that right column will become filled with more links and possibly some images, and I want that area to be entirely separate from the main copy of the site. This separation alerts viewers that the right column is where they need to go to find more information.

Finally, as a finishing touch, I’ll add the page’s images. Unless the client has an image to use, the image that you choose should “fit” within the scheme of the layout. In other words, try to find an appropriate image that reflects the colours you’ve chosen for the site. In this case, I chose to add a little humour to the layout with a stock photo of a geek. One reason I chose this photo is that the man in the image is looking straight at the viewer, which means that this image may catch the viewer’s eye before anything else on the site. Since this image is so important, I was happy that the colours in his shirt seemed close to the gold and blue contained in the site. Finally, his black glasses echoed the heavy black accents in the site’s headings. With all that said, I added a blue cast to the shadows and a yellow cast to the highlights of this photo in Photoshop to help it fit in more with the overall colour scheme.

I also added a tag line for containing the blog post tags and a date/time stamp so that viewers can see how recent the blog entries are. All these elements, as you’ll see below, add more “weight” and confusion to the site. This is just one more reason why you might try to keep the important and vital information as simple as possible—the viewer will have enough to deal with when they click on this site as it is. See Figure 12 below to examine the final result:

Figure 12: The final design, ready for client inspection.

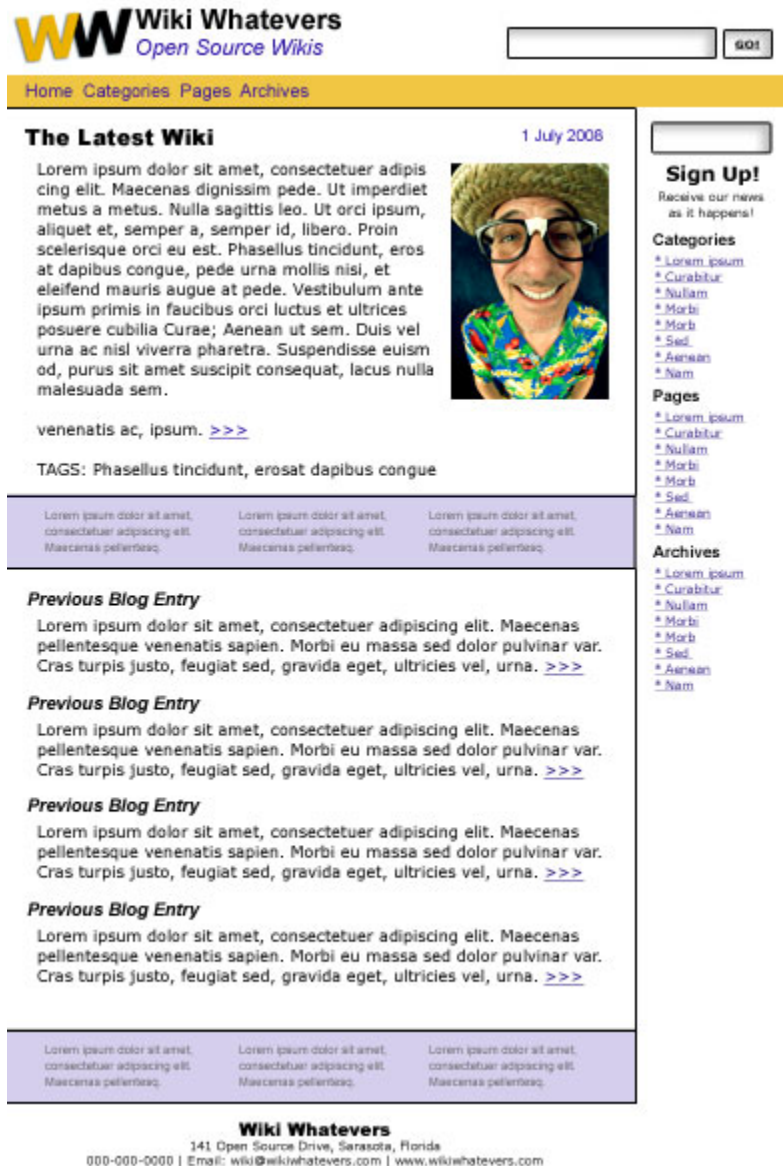
The nice thing about web design is that it isn't print design. Print is forever; the web constantly changes. So, this site may change over the years to reflect this group's growth. Mistakes can be corrected, and colours can be modified. With that said, it's best to have the most perfect product online from the get-go. This goal of offering the best product possible is good for your reputation and for your client's reputation as well.

As I build more pages, I'll take the finished pages on to the next and final step before they go to the client once again.

Fourth step: testing

Testing a web page means that the designer will go through all web pages with a fine-toothed comb to look for a variety of errors so they can be fixed before the site goes "live". You have several options for this testing, as several tests need to be conducted.

1. **Testing Typography and Links:** You can use friends, forums, and professional editors to check for typographical and spelling errors. While those folks are looking over the pages for you, ask them to check links as well to make sure they work. A word of warning—most clients would not like the idea that their copy is out in the public venue before it goes "live" on the Web. In this case, build the cost of an editor into your billing on the front end and have that editor sign a nondisclosure agreement ([NDA](#)) so that he/she is held responsible for keeping that web site's copy private.
2. **Test Code Validity:** Use the W3C validators to check [HTML](#) and [CSS](#) code every time you add new code to web pages. Without this step, the next step is a moot issue, as one code error can throw a web page design off in any number of browsers. You might learn along the way that any code you add for advertising will not validate. Don't try to change the code, however, as you may negate the value of the ads by creating errors in the ad code. Instead, most designers have learned to live with the fact that advertising code is the way it is, and not much can be done to change it. Thankfully, most advertising code will not affect your layout in the next step.
3. **Test Browser Compatibility:** You may wonder if you need to rush out and purchase several different computers and different monitor sizes along with several different operating systems to test a web site. You don't. Few web designers can afford that extravagance, so they use [several options](#) to test a web site's compatibility across browsers. These options include downloading several versions of one operating system on one machine, the advice from friends and/or forums, and the use of services



that provide screen captures. Screen capture services basically provide a screenshot of one web page at a time from several different machines. With this service you can see whether your type is too large for one resolution or too small for another. Or, you can discover whether your layout forces one user to scroll horizontally, or whether you've completely lost a column altogether in another browser. I use screen captures for testing in most cases, as the first two options have proven spotty at best. Plus, I don't like to send my clients' work out when I can keep their information private through the use of screen capture services. A few free web capture services exist, but I've learned that the queues are long and the options for a variety of testing environments may be limited. So, I spend a little money on [BrowserCam](#), and this service has proven worth its salt over several years. This service also provides a free trial, so you can test their product out at no charge to understand how it functions.

4. **Test for Usability and Accessibility:** You can find a number of [online tools](#) to test for accessibility. Some require the use of sound so that you can hear how a speech machine would "read" your page. Other test results may ask for a simple code change or a change in colours to provide more contrast for low-vision users. As far as usability, you can find tools and [checklists](#) online that will help you to make sure the site that you design is easy to use by the largest number of readers as well.

This testing is a tedious business, and you may discover in the process that your lovely design will stay lovely across some browsers, but may end up looking like last week's spaghetti warmed over in some of the others. The most important thing to remember is the content—if the content is visible and legible in all browsers, then that fraction of a space that's bothering you at the top of the header in that one browser isn't that important. If the majority of that site's users have no problem gaining access to the online materials within the site that you designed, then you've accomplished a goal that many designers neglect in favor of a design that they believe will win awards.

Summary

While colour and layout hold great appeal to the designer, other design elements must be dealt with as well. Typography, images, and the ability to blend in a client's needs for advertising and monetisation all play a part in web site design. The demands placed upon a designer to go with the flow of a client's desires, to meet the needs of a readership that demands accessible and usable sites, and still create a great design can seem overwhelming at times.

Even more frustrating is the lack of compatibility among different browsers. Although progress has been made over the past decade toward more compatibility, you must understand that the finished design may seem to take on a life of its own in various browsers. In addition, you should realise that users can change a web site with the click of a button in some browsers. Images can be eliminated, backgrounds and text colour changed, and anything with a hint of Javascript can be ignored by users.

On the other hand, the progress toward a more compatible environment and an exciting new era of web functionality can prove an exciting challenge to web designers today. And to think—it's been less than thirty years since home computers were widely available on the retail market. Think about what the next decade might bring to the designer who is willing to stay abreast of all the changes!

Exercise questions

- What are the four main types of fonts?
- Which fonts are best for body copy and why?
- Why is it important to create enough contrast between body copy and the background of that copy?
- Name at least two ways to break up a page filled with body copy.
- Give one reason why it's a good idea to add typography to a page before you add images.
- Name four types of alignment.
- Explain how alignment can help a web page look "cleaner".
- What is an NDA and when should you use that document?
- Why is it important to check the spelling on a web page?
- Name four ways to test a site before it goes "live".

About the author



Linda Goin carries a BFA in visual communications with a minor in business and marketing, and an MA in American History with a minor in the Reformation. While the latter degree doesn't seem to fit with the first educational experience, Linda has used her 25-year design expertise on site at archaeological digs and in the study of material culture.

Accolades for her work include fifteen first-place Colorado Press Association awards, numerous fine art and graphic design awards, and interviews about content development with The Wall St. Journal, Chicago Tribune, Psychology

Today, and LA Times. Linda is the author of several ebooks on web design, accessibility, and—as a sideline—also writes personal finance articles and ghostwrites for a few SEO pros.

11: Typography on the web

BY [PAUL-HAINE](#) · 8 JUL, 2008

Introduction

What is typography? Put simply, it is the art, design, and arrangement of text (referred to as type)—a concept borrowed from traditional print design. It is as much about what you *don't* do with your type as what you *do*. On the web, typography often gets very little attention, and there are certain technological limitations that can cause web typography to suffer when compared to print typography. However, with the tools available to you, there's no reason why type cannot be presented on the web in a wide variety of stylish and beautiful forms.

In this article I'll look at exactly why typography is limited on the web (compared to print design) and present some tips to follow for good web typography, along with an example web page that demonstrates some of these tips. Don't worry if you don't understand the CSS and HTML code at this stage—the point here is to make you think about design. While you are going through the article, it might be an idea to have a pen(cil) and paper by your side so you can start to sketch ideas about text layout. The table of contents for this article looks like so:

- [Limitations of web typography](#)
 - [Reduced selection of fonts](#)
 - [Hyphenation](#)
 - [Kerning](#)
 - [A lack of control](#)
- [How is typography done on the web?](#)
- [Quick tips](#)
 - [Select a range of fonts](#)
 - [Line length](#)
 - [Line height](#)
 - [Drop caps](#)
 - [Small caps](#)
 - [Hanging punctuation](#)
 - [Typographically-correct punctuation and other entities](#)
 - [Pull quotes](#)
- [Summary](#)
- [Exercise questions](#)

Limitations of web typography

Traditional print designers have a huge amount of options available to them when it comes to typography, including the sheer numbers of fonts available, and options for positioning text. Typography on the web is a lot more limited, because we must design using fonts and positioning, etc that we know will be available on the computers of the users that will look at their web sites—it is no use just designing for yourself on the web!

Limitations of web typography include:

- A reduced selection of fonts
- No hyphenation, making full justification look ugly when a column of text gets narrow
- Poor control over kerning (ie, the spacing between the letters)
- A lack of control over how the work is viewed—designers must account for a wide variety of screen sizes, resolutions and environments

Let's look at these points in a bit more depth.

Reduced selection of fonts

The reduced selection of fonts is often the first thing you will run up against when styling your text. Although you can specify any font you like in your CSS, visitors to your sites will only see your text displayed in that font if they happen to have it installed on their own computer—if they don't, their browser will either use an alternative font that you've specified in your CSS, or resort to the browser default (usually Times New Roman). So, you may like to see all your body text displayed with special fonts like Trump Medieval or Avant Garde, but unless your target audience is heavily biased towards designers your viewers likely aren't going to get the benefit. For this reason, most web designers limit themselves to the most commonly-available fonts across all systems, which are usually limited to the following:

- Andale Mono
- Times New Roman
- Georgia
- Verdana
- Arial/Arial Black
- Courier/Courier New
- Trebuchet MS
- Comic Sans (this is a terribly unprofessional, ugly font—don't use this)
- Impact

These look like Figure 1:

Figure 1: The most commonly available fonts across all systems are limited to the above.

Specifying any of the above fonts means you're reasonably likely to be picking a font that most of your visitors will also have. Microsoft also introduced six new fonts designed for screen use in Windows Vista and XP, and, oddly, chose to begin all their names with the letter C. If you want to use them, they are Cambria, Calibri, Candara, Consolas, Constantia and Corbel. I'd advise against using these, however, because they are not likely to be available on the Mac or Linux platforms.

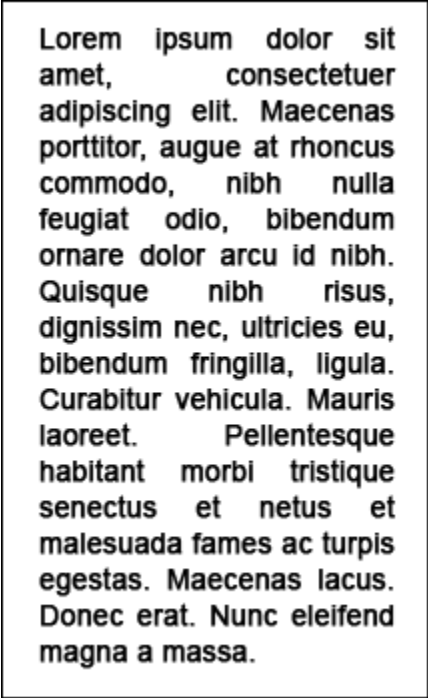
So, compared to the thousands of typefaces available to print designers, web designers can reliably choose from just over a dozen. But is this a serious limitation? Typography is about far more than simply selecting an attractive font, it's about line lengths and kerning and white space as well—remember that typographers pre-dating electronic fonts would have faced similar limitations.

Hyphenation

When it comes to aligning your text within its container, there are four options: left-aligned, right-aligned, centre-aligned and fully-justified.

Andale Mono
Times New Roman
Georgia
Verdana
Arial
Courier New
Trebuchet MS
Comic Sans
Impact

Fully-justified text, where both the left and right edges of the block are aligned to the vertical sides of their container, can look more attractive than text with a “ragged” edge, and you’ll see it a lot in magazines and books. On the web, however, it’s problematic due to the lack of automatic hyphenation, which breaks words at appropriate points to better fit them in the line. To fully justify the block of text, all the browser can do is adjust the spacing between the words, which can lead to “rivers of white space” running vertically through the block—this usually happens when the line length within the block is too short and there aren’t enough spaces to adjust subtly, as shown in Figure 2.



Lorem ipsum dolor sit
amet, consectetur
adipiscing elit. Maecenas
porttitor, augue at rhoncus
commodo, nibh nulla
feugiat odio, bibendum
ornare dolor arcu id nibh.
Quisque nibh risus,
dignissim nec, ultricies eu,
bibendum fringilla, ligula.
Curabitur vehicula. Mauris
laoreet. Pellentesque
habitant morbi tristique
senectus et netus et
malesuada fames ac turpis
egestas. Maecenas lacus.
Donec erat. Nunc eleifend
magna a massa.

Figure 2: Rivers of whitespace can spoil justified text blocks.

As you can see in this screenshot, the lack of hyphenation to break words at natural points has caused the spacing between certain words to grow to unacceptable sizes. To avoid this, you should use left-aligned text for the most part on the Web.

Kerning

Kerning is the process of adjusting the spacing between particular pairs of letters when the font in use is a proportional one (such as Times New Roman, where the space between each character varies from character to character) rather than a monospaced one (such as Courier, where the space between each character is the same each time). It’s used in print to tighten up the space between letters that align naturally, such as a W followed by an A, and can give a more professional look and feel to the text. Most professional fonts come with kerning instructions built in, to provide spacing information to the type renderer. See Figure 3 for an illustration of the difference kerning makes.

Figure 3: Kerning can certainly improve the look of text.

In the above screenshot, the first word has not been kerned. The second word, though, has had the spacing between the W and the A reduced, whilst the space between the A and the S has been increased a touch.

On the web, kerning with this level of precision is effectively unavailable. The only thing we have that comes close to it is the ability to use tracking, which in the print world means adjusting the space between characters throughout the copy, no matter what those characters are—so, you could decrease the space between your W and your A, but you’ll also be affecting the space between every other letter. On the web, tracking is better known as letter spacing, and is controlled with CSS—this is illustrated in Figure 4.



WASP
WASP



WASP

Figure 4: Proper kerning is not available on the Web; the closest we have available is more general letter spacing.

In the above screenshot, the spacing between each character has been increased by the same amount. Whilst

this has helped separate the A and the S, the space between the W and A is now too much. Letter spacing with CSS is a difficult property to use effectively due to this all-or-nothing nature, and for this reason it is best used sparingly.

A lack of control

With all this talk of the print world, there's something very important worth bearing in mind, and that is that *the web is not print*. So where the print designer doesn't have to worry about the end viewer resizing the text, or not having the desired set of fonts, or not having aliasing enabled, we do, and there's often a temptation to try and force a particular design upon the viewer—fixing a rigid text size for instance, or placing text in fixed-width and fixed-height containers, or even replacing whole chunks of text with images.

This lack of control needn't be a problem however—you just have to get used to the idea that people will want to read your content on a variety of devices in a variety of environments in a variety of ways. You shouldn't try to stop them, or make it difficult for them—if they want to read your content then it should be as easy to do so as possible. They may wish to read it on their mobile device during their commute home; they may prefer to print everything out and read it on paper instead of a screen; they may be visually impaired and need to increase the font size somewhat. This is why, when you style your text on the web, what you're really doing is providing a guide to all the different browsing devices as to how you'd prefer that text be seen. Devices are free to ignore everything you say, of course, but that's ok—what matters is that you're not trying to force your design decisions on your entire audience.

How is typography done on the web?

Typography on the web is controlled entirely with CSS, and by using CSS you gain a lot of control: not just over the size, colour and typeface selection but also over the line height, the letter spacing, the level of capitalisation (all caps, initial caps, small-caps or no capitalisation at all) and even control over how the first letter or first line of your text is styled.

By styling the text's containing block, you also have control over the level of justification of the text and the line length. Not only that, you also only have to create your style rules in one location—your stylesheet—to have those rules apply to all of your text, across your whole website (or you can be specific and target particular paragraphs, or areas on the page). Furthermore, if you ever find yourself needing to increase your website text size, or change the body font, you only have to change the value in your stylesheet.

Quick tips

Here are some quick tips to help you out with typography on the Web.

Select a range of fonts

It's good practice to include back-up selections when specifying your preferred display font. So, rather than simply specifying "Georgia", you could specify "Georgia, Cambria, 'Times New Roman', Times, serif". So, first the browser will attempt to use a font named Georgia, but if this font isn't installed it will try for Cambria, then Times New Roman, followed by Times, followed by whatever the operating system has assigned to the "serif" keyword.

Line length

To aid readability, the average length of a line of text within your containing block should be around 40-60 characters per line, though this should vary depending on your audience (children prefer shorter line lengths, adults prefer longer).

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas porttitor, augue at rhoncus commodo, nibh nulla feugiat odio, bibendum ornare dolor arcu id nibh. Quisque nibh risus, dignissim nec, ultricies eu, bibendum fringilla, ligula. Curabitur vehicula. Mauris laoreet. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Maecenas lacus. Donec erat. Nunc eleifend magna a massa.

An ideal line length is shown in Figure 5:

Figure 5: 60 characters per line—the ideal line length.

The text in the screenshot is about 60 characters per line. Any more than this and the reader may have to start moving their eyes—or even their head—in order to follow the text, which can increase eye-strain and makes the text harder to take in.

Line height

Line height refers to the vertical space between your lines, and you can make your type more readable by increasing it a little above the browser default (which also allows more space for subscript and superscript characters)—see the difference between the two paragraphs in Figure 6:

Figure 6: Line height can make a big difference to the look and feel of text.

The first paragraph in the above screenshot has a default line height, and can feel a little cramped. The second paragraph has had its line height increased, and the text has a bit more room to breathe, making it a bit more readable. Too much line height, though, and you make the text harder to read again, so be careful.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas porttitor, augue at rhoncus commodo, nibh nulla feugiat odio, bibendum ornare dolor arcu id nibh. Quisque nibh risus, dignissim nec, ultricies eu, bibendum fringilla, ligula.

Curabitur vehicula. Mauris laoreet. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Maecenas lacus. Donec erat. Nunc eleifend magna a massa.

Drop caps

By targeting the `first-letter` pseudo-element with something like `p:firstletter { }`, you can style the first letter of a line differently from the rest—such styling is usually known as a drop cap, where the first letter takes up about 3-4 lines of text—see Figure 7.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas porttitor, augue at rhoncus commodo, nibh nulla feugiat odio, bibendum ornare dolor arcu id nibh. Quisque nibh risus, dignissim nec, ultricies eu, bibendum fringilla, ligula.

Figure 7: A typical drop cap.

Small caps

Often, fonts come with a small-caps variation—a set of capitalised letters that are uppercased but approximately the size of the lowercase variant. This is useful for

when you want to capitalise something but don't want to draw too much attention to it, so it can be used for abbreviations, for example. Even if the system doesn't have a small-cap variant of the specified font, that's ok—the browser will generate its own version by using full capitalisation and then shrinking the characters to around 70-80% of their normal size. Figure 8 shows small caps in action.

Figure 8: Small caps in action.

Hanging punctuation

ABANDON HOPE
ALL YE WHO ENTER HERE

A good typographical effect can be used if your sentence starts with quote marks. Using the `text-indent` CSS property combined with a negative value—either a value in ems (-10em), points (-10pt), pixels (-10px) or percent (-10%)—allows you to shunt the quote mark out into the left, maintaining the left vertical line of your block of text, as shown in Figure 9:

Figure 9: Hanging punctuation.

Typographically-correct punctuation and other entities

You can make your text look more professional and elegant by using the wide variety of typographic HTML entities that are available such as “smart” or “curly” quotes and en- and em-dashes. A lot of blogging and word processing software will automatically do this for you as you type, turning your regular straight quotes into the typographically-correct curly variety, and turning strings of dashes into en and em dashes. See Figure 10 for examples of typographically-correct punctuation.

**“Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Maecenas porttitor, augue at rhoncus commodo, nibh
nulla feugiat odio, bibendum ornare dolor arcu id nibh.”**

**“Quisque nibh risus, dignissim nec, ultricies eu, bibendum
fringilla, ligula.”**

**“This paragraph
not only has
curly quotes but
also — yes! —
typographically-
correct dashes
as well.”**

Figure 10: Typographically-correct punctuation

Once you start peppering your copy with smart punctuation, your text can look far more elegant and professional—more like something from a magazine or a book than from online. Bear in mind though that this sort of punctuation can look a little pixellated for people with older screens or with aliasing disabled, so use with caution.

Then there’s entities—bits of special HTML that you can insert into your copy to generate special characters not easily available from your keyboard. Figure 11 contains a number of entities:

Figure 11: HTML entities

These can be typed in by hand, but a lot of content management software can convert or insert these for you with ease.

Pull-quotes

A pull-quote is a short extract from your text that appears elsewhere on your page with a larger text size, and sometimes a different font, to draw attention to it. You’ll have seen them in almost every magazine you’ve ever read, and they’re an effective way of breaking up your text and highlighting key quotes or phrases—and they’re also easy to do on the web with some simple markup and styling. Just make the text

larger, perhaps set it in a different font, position it so that the regular text wraps around it and you're done. There are also some more advanced solutions that involve JavaScript picking out selected text and automatically populating a pull quote from it, which can save you having to write the same text twice in your markup.

How many entities can we fit in one paragraph? Well, we could show you a TM symbol, an ® symbol, a © symbol, a • point, a € currency symbol, a ¥ symbol...and there goes a typographically-correct ellipsis!

And dón't forgét äcçented çharacters, either.

Summary

So that's typography, and typography on the web; hopefully you can see that text online needn't be limited to Verdana, small, #333333—there is a wide range of typographic tricks and tips that can help make your text stand out from the rest of the crowd. For most websites, the reason people will be visiting is to read what you or your authors have written; it makes sense, then, to make that reading as pleasurable as possible.

Exercise Questions

- What's the difference between kerning and tracking, and which one is available to the web designer?
 - How can you avoid "rivers of white space" running through your text?
 - Name the four different types of capitalisation available through CSS.
 - What's a good line length for body content, and what factors can affect it?
 - What's the difference between a serif font and a sans-serif font? Give an example of each.
- How does hanging punctuation differ from regular punctuation?
 - If you want to insert a copyright symbol into your copy, you use an HTML entity. Have a look on the internet and see if you can find all the other HTML entities. There's about 250 of them!

About the author

Clawing his way from deepest, darkest Somerset upon his coming of age, Paul Haine found himself ironically trapped for a further six years on the opposite side of the country in deepest, darkest Kent, learning about web standards during the spare weeks between history lectures. After spending two years in Oxford and writing [HTML Mastery](#), he moved to London's Famous Islington where he works as a client-side developer for [The Guardian](#).

Paul's personal website, <http://joeblade.com>, is missing, presumed dead.

12: The basics of HTML

BY [MNFRANCIS](#) · 8 JUL, 2008

Introduction

In this article you will learn the basics of HTML—what it is, what it does, its history in brief, and what the structure of an HTML document looks like. The articles that follow this one will look at each individual part of HTML in much greater depth. The structure of this article is as follows:

- [What HTML is](#)
- [What HTML looks like](#)
- [The history of HTML](#)
- [The structure of an HTML document](#)
- [The syntax of HTML elements](#)
- [Block level and inline elements](#)
- [Character references](#)
- [Summary](#)

What HTML is

Most desktop applications that read and write files use a special file format. For example, Microsoft Word understands “.doc” files and Microsoft Excel understands “.xls”. These files contain the instructions on how to rebuild the document the next time you open it, what the contents of that document are, and “metadata” about the article such as the author, the date the document was last modified, even things such a list of changes made so you can go back and forth between versions.

HTML (“HyperText Markup Language”) is a language to describe the contents of web documents. It uses a special syntax containing markers (called “elements”) which are wrapped around the text within the document to indicate how user agents should interpret that portion of the document.

The technical term “user agents” is used here rather than “web browsers”. A user agent is any software that is used to access web pages on behalf of users. There is an important distinction to be made here—all types of desktop browser software (Internet Explorer, Opera, Firefox, Safari, etc.) and alternative browsers for other devices (such as the Wii Internet channel, and mobile phone browsers such as Opera Mini and WebKit on the iPhone) are user agents, but not all user agents are browser software. The automated programs that Google and Yahoo! use to index the web to use in their search engines are also user agents, but no human being is controlling them directly.

What HTML looks like

HTML is just a plain textual representation of content and its general meaning. For example, the code for the “The Purpose of HTML” header above looks like:

```
<h2 id="htmllooks">What HTML looks like</h2>
```

The “<h2>” part is a marker (which we refer to as a “tag”) that means “what follows should be considered a second level heading”. The “</h2>” is a tag to indicate where the end of the second level heading is (which we refer to as a “closing tag”). The opening tag, closing tag and everything in between is called an “element”. Many people use the terms element and tag interchangeably however, which is not strictly correct. The `id="htmllooks"` is an attribute; you'll learn more about these later on.

In most browsers there is a “Source” or “View Source” option, commonly under the “View” menu. If you have the option, choose it now and spend some time looking at the HTML source for this page.

The history of HTML

In the article [The history of the internet and the web](#) you learned a little about how the modern Web came about. When Tim Berners-Lee invented the World Wide Web, he created both the first web server and web browser and [the first version of HTML](#).

Whilst HTML has changed considerably since the first days, a lot of the content of modern-day HTML is embodied in that first documentation and more than half of the “tags” described in the original “HTML tags” document still exist.

As more people started writing web pages and alternatives to the original browser software, more features were being added to HTML. Many were adopted universally (such as the `img` element used to insert an image into a document, first implemented in NCSA Mosaic). Some were more proprietary and really only used in one or two browsers. There was a growing need for standardisation—so that authors of other web browsing software had a document (called a “specification”) that definitively described to them what HTML looked like so they could judge whether or not they were missing out on implementing some parts of HTML.

The IETF (Internet Engineering Task Force—a standards body concerned with inter-operability across the internet) published a draft proposal of HTML in 1993. This expired without becoming a standard in 1994, but prompted the IETF to create a working group to look at HTML standardisation.

In 1995, “HTML 2.0” was written, taking ideas from the original HTML draft. An alternate proposal called HTML+ was also written by Dave Raggett, which was used as a basis for many of the new elements implemented by browsers (such as the method for inserting images into documents, pioneered by NCSA Mosaic).

A draft of HTML 3.0 followed later that year, but work on that version was discontinued because of a lack of support for the direction from browser makers. HTML 3.2 dropped many of the new features of 3.0, and instead adopted many of the creations of the then-popular browsers Mosaic and Netscape Navigator.

In 1997, the W3C published HTML 4.0 as a recommendation that adopted more browser-specific extensions but also attempted to rationalise and clean up HTML. This was done by marking various elements as deprecated—which means the elements are obsolete and whilst they still exist in this version they will be removed in a later revision. This was to encourage better and more semantic use of HTML in documents (described in more detail in [The web standards model](#) article).

HTML 4.01 was published in 1999, with some errata noted in 2001. This is the latest version of HTML, although HTML 5 is currently being drafted.

In 2000, the W3C also published the XHTML 1.0 specification, which was HTML re-structured to be a valid XML document.

The structure of an HTML document

The smallest valid HTML document possible would be something like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
  <html>
    <head>
      <title>Example page</title>
    </head>
    <body>
      <h1>Hello world</h1>
    </body>
  </html>
```

The document first starts with a document type element, or doctype (described in more detail in the [Choosing the right doctype...](#)). This describes which type of HTML is being used—so that user agents can

determine how to interpret the document, and work out whether it is following the rules it says it is going to follow.

After this, you can see the opening tag of the `html` element. This is a wrapper around the entire document. The closing `html` tag is the last thing in any HTML document.

Inside the `html` element, there is the `head` element. This is a wrapper to contain information about the document (the metadata). This is described in more detail in [the next article](#). Inside the `head` is the `title` element, which defines the “Example page” heading in the menu bar.

After the `head` element there is a `body` element, which is the wrapper to contain the actual content of the page—in this case, only a level-one header (`h1`) element, which contains the text “Hello world.” And that’s our document in full.

As you can see, elements often contain other elements. The body of the document will invariably end up involving many nested elements. Page divisions create the overall structure of the document, and will contain subdivisions. These will contain headings, paragraphs, lists and so on. Paragraphs can contain elements that make links to other documents, quotes, emphasis and so on. You will find out more about these elements as the series progresses.

The syntax of HTML elements

As you have already seen, a basic element in HTML consists of two markers around a block of text. There are some elements that do not wrap around text, and in almost every case elements can contain sub-elements (such as `html` containing `head` and `body` in the example above).

Elements can also have attributes, which can modify the behaviour of the element and introduce extra meaning.

```
<div id="masthead">
  <h1>The Basics of
    <abbr title="Hypertext Markup Language">HTML</abbr>
  </h1>
</div>
```

In this example a `div` element (page division, a way of breaking documents into logical blocks) has had an `id` attribute added, and this has been given a value of `masthead`. The `div` contains an `h1` element (first, or most important level header), which in turn contains some text. Part of that text is wrapped in an `abbr` element (used to specify the expansion of abbreviations), which has the attribute of `title`, the value of which is set to `Hypertext Markup Language`.

Many attributes in HTML are common to all elements, though some are specific to a given element or elements. They are always of the form `keyword="value"`. The value should be surrounded by single or double quotes (in some circumstances the quotes can be left out, but this is not a good practice in terms of predictability, understanding and clarity—you should *always* put quotes around your values).

[Attributes and their possible values are mostly defined by the HTML specifications](#)—you cannot make up your own attributes without making the HTML invalid, as this can confuse user agents and cause problems interpreting the web page correctly. The only real exceptions are the `id` and `class` attributes—their values are entirely under your control, as they are for adding your own meaning and semantics to your documents.

An element within another element is referred to as being a “child” of that element. So in the above example, `abbr` is a child of the `h1`, which is itself a child of the `div`. Conversely, the `div` would be referred to as a “parent” of the `h1`. This parent/child concept is important, as it forms the basis of CSS and is heavily used in JavaScript.

Block level and inline elements

There are two general categories of elements in HTML, which correspond to the types of content and structure those elements represent—block level elements and inline elements.

Block level means a higher level element, normally informing the structure of the document. It may help to think of block level elements being those that start on a new line, breaking away from what went before. Some common block level elements include paragraphs, list items, headings and tables.

Inline elements are those that are contained within block level structural elements and surround only small parts of the document's content, not entire paragraphs and groupings of content. An inline element will not cause a new line to appear in the document, they are the kind of elements that would appear in a paragraph of text. Some common inline elements include hypertext links, highlighted words or phrases and short quotations.

Character references

One last item to mention in an HTML document is how to include special characters. In HTML the characters `<`, `>` and `&` are special. They start and end parts of the HTML document, rather than representing the characters less-than, greater-than and ampersand.

One of the earliest mistakes a web author can make is to use an ampersand in a document and then have something unexpected appear. For example, writing "Imperial units measure weight in stones£s" could actually end up appearing as "...stones&s" in some browsers.

This is because the literal string "`£`" is actually a character reference in HTML. A character reference is a way of including a character into a document that is difficult or impossible to enter using a keyboard, or in a particular document encoding.

The ampersand (`&`) introduces the reference and the semi-colon (`;`) ends it. However, many user agents can be quite forgiving of HTML mistakes such as leaving out the semi-colon, and treat "`£`" as a character reference. References can either be numbers (numeric references) or shorthand words (entity references).

An actual ampersand has to be entered into a document as "`&`", which is the character entity reference, or as "`&`", which is the numeric reference. [A full chart of character references can be found on evolt.org](http://evolt.org).

Summary

In this article, you have learned the basics of HTML, where it has evolved from and have some insight into the structure of an HTML document. We will now continue to describe the `<head>` section of an HTML document in some more detail, before continuing to address the `<body>` content.

About the author



Photo credit: [Andy Budd](#).

Mark Norman Francis has been working with the internet since before the web was invented. He currently works at Yahoo! as a Front End Architect for the world's biggest website, defining best practices, coding standards and quality in web development internationally.

Previous to Yahoo! he worked at Formula One Management, Purple Interactive and City University in various roles including web development, backend CGI programming and systems architecture. He pretends to blog at <http://marknormanfrancis.com/>.

13: The HTML <head> element

BY [CHRISTIAN HEILMANN](#) · 8 JUL, 2008

Introduction

This article deals with a part of the HTML document that does not get the attention it deserves: the markup that goes inside the `head` element. By the end of this tutorial you'll have learnt about the different parts of this section and what they all do, including the `doctype`, `title` element, keywords and description (which are handled by `meta` elements). You'll also learn about JavaScript and CSS styles (both internal and external) and about what not to leave in the `head`. You can [download some demo files here](#), which I will refer to during the article; feel free to play with them however you like once you have finished working through it. Make sure you go through the full tutorial from start to finish, as it builds up a series of best practices to follow when dealing with the HTML `head`. Whilst each part is valid in itself, there is a conclusion at the end about the best practices that will make you reconsider some of the earlier advice. The contents are as follows:

- [Head? What head are we talking about?](#)
- [Setting the document's primary language](#)
- [Judging a document by its title](#)
- [Adding keywords and a description](#)
- [What about the looks? Adding styles](#)
- [Adding dynamic features using JavaScript](#)
- [Stop right there! Using inline CSS and JavaScript is not the greatest idea!](#)
- [Summary](#)
- [Exercise questions](#)

Head? What head are we talking about?

Earlier on in this course you learnt that a valid HTML document needs a `doctype`—the `doctype` explains what type of HTML is to be expected and instructs browsers to show the document accordingly. [Roger goes into the doctype in much more detail in Article 14](#), but for now, let's just say that the `doctype` specifies that the document needs an `html` element with `head` and `body` elements inside it. The `body` is where you'll spend most of your time, as this is where all the content of the document goes. The `head` plays a seemingly less important role, because with the exception of the `title` element, nothing you put in this section is directly visible to your site's visitors. Instead, the `head` is the place where most of the instructions for the browser happen and where you store extra information—so called `meta` information—about the document.

Setting the document's primary language

One piece of information about the document goes on the parent of the `head`, the `html` element. It is here that we define the primary natural language of the document. By natural language, I mean *human* language, such as French, Thai or even English. This helps screenreaders, because the word "six" is pronounced very differently in French and English, and may also help search engines.

The attributes you use to set the language depend on the `DOCTYPE` of your document. The [W3C says](#)

For HTML use the `lang` attribute only, for XHTML 1.0 served as text/html use the `lang` and `xml:lang` attributes, and for XHTML served as XML use the `xml:lang` attribute only.

The language codes may be two-letter codes, such as `en` for English, four-letter codes such as `en-US` for American English, or other, less common, codes. The two-letter codes are defined in [ISO 639-1](#).

Judging a document by its title

One of the most important elements in the `head` is the `title`. The text contained within the `title` is displayed by pretty much all user agents/browsers in the browser application title bar (the bar bordering the top of the browser window). It is the first piece of content that web users will see when they visit your site, and therefore it is very important. In addition, assistive technologies like screen readers (software that reads out web pages to users with visual impairments) give this as a first hint of what visitors can expect from the document, and a lot of search engines work similarly too, so your chances to get found on the web increase drastically when you use a good title that is human readable and contains the right keywords. Let's take the following HTML document ([headexample.html](#) in your zip file) and open it in a browser.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>I am a title example</title>
</head>
<body>
</body>
</html>
```

You'll see that the text in the `title` is displayed in the bar above the browser navigation as shown in Figure 1.

Figure 1: Displaying a title in a browser.

There are many tutorials on the Web about how to write good document titles, most of which are related to search engine optimization (SEO). Don't get overboard and try to trick the search engines into showing up an inflated number of search results. Write a concise short information piece about what the document is. "Breeding Dogs—Tips about Alsatians" is a lot more human digestible than "Dogs, Alsatian, Breeding, Dog, Tips, Free, Pet."

Adding keywords and a description

The next thing to do might seem superfluous at first as it will never be directly visible to your visitors: adding a description and keywords. Both of these get added to the `head` inside `meta` elements, as shown in the following example taken from the Yahoo! Eurosport site ([headwithmeta.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Yahoo! UK & Ireland Eurosport'Sports News | Live Scores | Sport</title>
  <meta name="description" content="Latest sports news and live scores from
Yahoo! Eurosport UK. Complete sport coverage with Football results, Cricket
scores, F1, Golf, Rugby, Tennis and more.">
  <meta name="keywords" content="eurosport,sports,sport,sports news, live
scores, football, cricket, f1, golf, rugby, tennis, uk, yahoo">
</head>
<body>
</body>
</html>
```

If you open this document in a browser you'll not see anything in the `body` of the document, however, if you put this document online and search engines index it, the description will be displayed as the text below the link in search engine results, as shown in Figure 2.

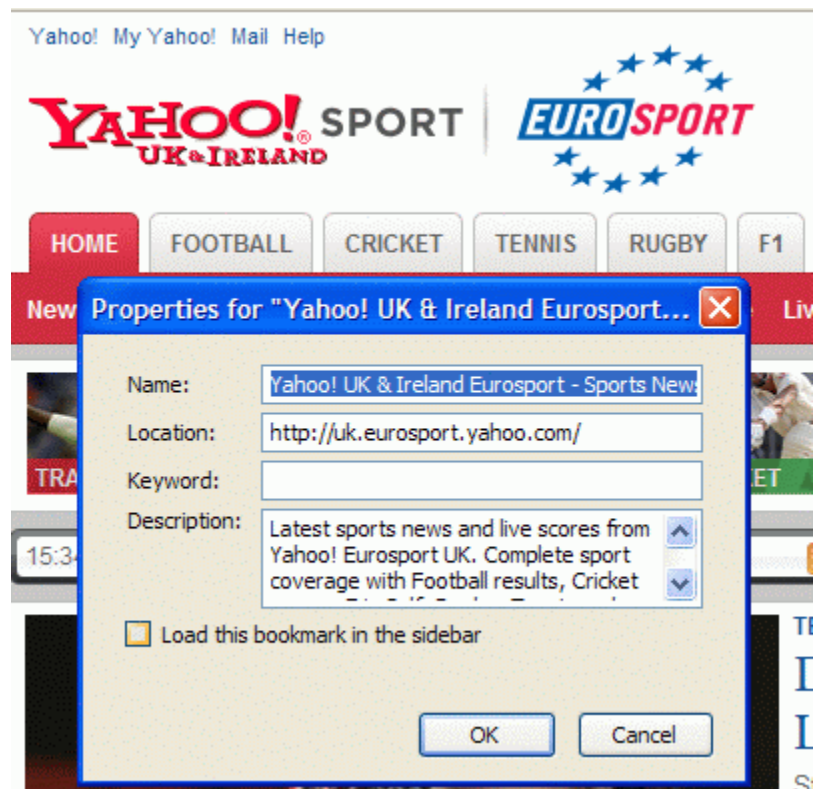
Figure 2: Descriptions show up in search engine result pages.

This could be the crucial bit of information a possible visitor of your site was looking for and the reason for him or her to click the link. Descriptions have another use too—some browsers show the description as extra information when you add the document to your favourites, as shown in Figure 3:

Figure 3: Descriptions show up in some browsers when you add the document as a favourite.

So although there is no obvious immediate benefit to adding meta descriptions, they do play an important role in the success of your document. The same—to a smaller degree—applies to the keywords you added.

Although years of abuse by spammers made search engines not take keywords seriously any more, they can however still be a really good tool to use if you want to quickly index a lot of documents without having to scan and analyze the content. You could use the meta keywords for example in a content management system by writing a script that indexes them and makes your search engine a lot faster. It doesn't hurt to provide a way of finding documents without having to analyze their content. By adding some keywords in a meta element you leave yourself the option to have a clever and fast search for your sites should you want to create something like this in the future. Think of keywords as small bookmarks you leave in a massive book to make it easier for yourself to find a certain section immediately without having to read through whole chapters.



What about the looks? Adding styles

The next thing you can add to the `head` of a document is style rules, defined in Cascading Style Sheets (CSS). You can embed those directly in the `head` using a `style` element, for example ([headinlinestyles.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Breeding Dogs—Tips about Alsatians</title>
```



```

<meta name="description" content="How to breed Alsatians, tips on proper
breeding and information about common issues with this breed.">
<meta name="keywords" content="Dogs,Alsatian,Breeding,Dog,Tips,Free,Pet">
<style type="text/css">
  body{
    background:#000;
    color:#ccc;
    font-family: helvetica, arial, sans-serif;
  }
</style>
</head>
<body>
<p>Test!</p>
</body>
</html>

```

If you open this in a browser it'll show the "Test!" text in grey on a black background, and the font will be Helvetica or Arial, depending what your system has installed. The `style` element can also contain another attribute called `media`, which defines what kind of media will use these styles, ie do you want these styles used when viewing the document on a computer screen, on a handheld device, or when printed out? There are several media types to choose from, the most useful being:

- `screen`—for displays on monitors.
- `print`—to define what a printout of the document should look like.
- `handheld`—to define the look and feel of the document on mobile devices and other handheld devices.
- `projection`—for presentations done in HTML, for example supported by the [Opera Show feature](#).

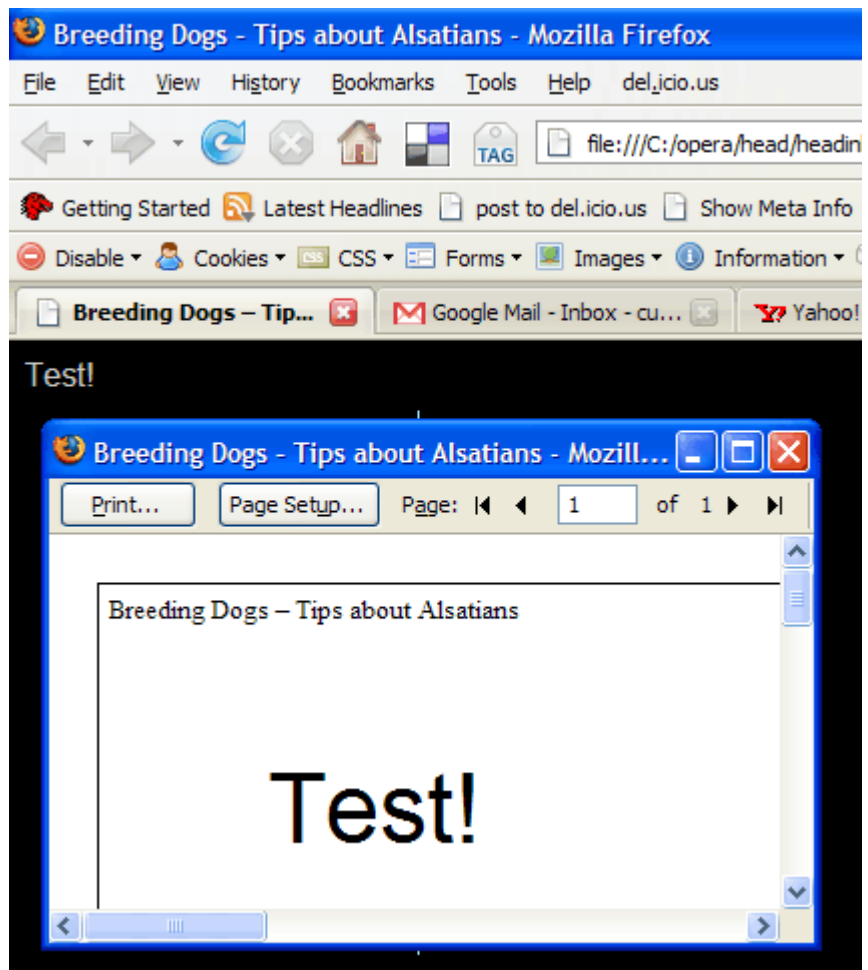
If you for example want to override the colours used on the screen and use a larger font size to make the page better for printing, you can add another style block below the first one, with a `media` attribute of `print`, as seen below (check out the full code in [headinlinestylesmedia.html](#)):

```

<style type="text/css"
media="print">
  body{
    background:#fff;
    color:#000;
    font-family:
helvetica, arial, sans-
serif;
    font-size:300%;
  }
</style>

```

Now when you go to print this web page, the browser knows to use the print stylesheet for printing the document, and not the screen



styles. Check this out by loading headinlinestylesmedia.html and selecting print preview, as shown in Figure 4:

Figure 4: A screen and a print style sheet.

Adding dynamic features using JavaScript

Another thing you can add to the `head` is scripts that get executed by the browser—so called “client side scripts”—written in JavaScript. As you learned in [Article 4](#), JavaScript adds dynamic behavior to the static HTML document, for example animation effects, or form data validation, or other things being triggered when the user performs certain actions.

You add JavaScript to a document using the `script` tag. When a browser encounters one of these, it’ll drop everything else and pause parsing of the rest of the document while it tries to execute the code inside it. This means that when you want to make sure that your JavaScript is available before the main document loads, you need to add it to the `head`. For example, you can give the visitor a warning that a certain link will take them to another server with the following script ([headscript.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Breeding Dogs—Tips about Alsatians</title>
  <meta name="description" content="How to breed Alsatians, tips on proper
breeding and information about common issues with this breed.">
  <meta name="keywords" content="Dogs,Alsatian,Breeding,Dog,Tips,Free,Pet">
  <style type="text/css" media="screen">
    body{
      background:#000;
      color:#ccc;
      font-family: helvetica, arial, sans-serif;
    }
    a {color:#fff}
  </style>
  <style type="text/css" media="print">
    body{
      background:#fff;
      color:#000;
      font-family: helvetica, arial, sans-serif;
      font-size:300%;
    }
  </style>
  <script>
    function leave(){
      return confirm("This will take you to another site,\n are you sure you want
to go?")
    }
  </script>
</head>
<body>
Test!
<a href="http://dailypuppy.com" onclick="return leave()">The Daily Puppy</a>
</body>
</html>
```

If you open this example in your web browser and click the link you’ll get asked to confirm your action. This is just a quick example of scripting and is far from what is best practice these days. Other tutorials later on in this course will deal with JavaScript best practices and teach you JavaScript techniques in depth, but don’t worry about it for now.

Stop right there! Inline CSS and JavaScript is not too clever!

Strong words, I know, but there is one thing you need to remember when you build web sites: the best thing to do is to re-use your code as much as possible. Adding site-wide styles and scripts into each and every one of your pages doesn't make much sense—on the contrary, it makes it harder to maintain a complete site and bloats the individual documents unnecessarily.

It is much better to put your styles and scripts in external files, and import them into your HTML files where needed, so you only need to update them in one place if changes need to be made. For your JavaScript, you do this using `script` elements that have no script inside them, but instead link to an external file using a `src` attribute, as seen in the code below ([externaljs.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Breeding Dogs—Tips about Alsatians</title>
  <meta name="description" content="How to breed Alsatians, tips on proper
breeding and information about common issues with this breed.">
  <meta name="keywords" content="Dogs,Alsatian,Breeding,Dog,Tips,Free,Pet">
  <style type="text/css" media="screen">
    body{
      background:#000;
      color:#ccc;
      font-family: helvetica, arial, sans-serif;
    }
    a {color:#fff}
  </style>
  <style type="text/css" media="print">
    body{
      background:#fff;
      color:#000;
      font-family: helvetica, arial, sans-serif;
      font-size:300%;
    }
  </style>
  <script src="leaving.js"></script>
</head>
<body>
Test!
<a href="http://dailypuppy.com" onclick="return leave()">The Daily Puppy</a>
</body>
</html>
```

It is not as easy with CSS. The `style` element does not have a `src` attribute, so you need to use the `link` element instead—it has an `href` attribute that specifies an external CSS file to import, and a `media` attribute to define if these styles should be used for screen, print etc, similar to what we saw earlier on. By putting both CSS and JavaScript into their own files you can cut down the length of the head immensely, as shown below ([externalall.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Breeding DogsTips about Alsatians</title>
  <meta name="description" content="How to breed Alsatians, tips on proper
breeding and information about common issues with this breed.">
  <meta name="keywords" content="Dogs,Alsatian,Breeding,Dog,Tips,Free,Pet">
  <link rel="stylesheet" type="text/css" media="screen" href="styles.css">
  <link rel="stylesheet" type="text/css" media="print" href="printstyles.css">
  <script src="leaving.js"></script>
```

```
</head>
<body>
Test!
<a href="http://dailypuppy.com" onclick="return leave()">The Daily Puppy</a>
</body>
</html>
```

The other benefits of keeping styles and scripts in their own files are:

1. You make it both faster and easier for your site visitors, because although a few extra files are downloaded, the style and script information doesn't need to be repeated in each web page file (it is just downloaded once, in the separate script/style files) so each page file downloaded will be smaller. In addition, the CSS and JavaScript files will be cached (stored temporarily on your local machine), so that the next time you access the site, the files will be on your computer already, meaning they don't need to be downloaded again.
2. Next comes ease of maintenance. The style and script for the whole site—which could be thousands of documents—are in one single location, so if you need to change something you only need to change one file, and not thousands.

Summary

That's it for this article. You've learnt about all the different parts that can be in the head section of HTML documents. They are:

- The `title`, which introduces the document.
- `meta` elements, which contain a description of the content of this document and keywords allowing for easier indexing of the content.
- `link` elements, which point to external CSS files.
- `script` elements that point to external JavaScript files.

Making sure that all of these are correct will result in your document being fast, and easy to find and understand.

Exercise questions

As usual, here are some exercise questions to test your comprehension of the subject area.

- Why does it make sense to add a description in a `meta` element if it doesn't get displayed on the screen?
- What is the benefit of adding JavaScript to the `head` of a document and not in the `body`?
- How can you benefit from your browser's caching and what do you need to do to make it work for you?
- As search engines give the title of a document much love, wouldn't it be useful to cram it full of relevant keywords? What are the downsides of this practice?
- As the title display can be a bit boring, wouldn't it make sense to bold some words with a `b` element? Is that possible?

About the author



Photo credit: [Bluesmoon](#)

Chris Heilmann has been a web developer for ten years, after dabbling in radio journalism. He works for Yahoo! in the UK as trainer and lead developer, and oversees the code quality on the front end for Europe and Asia.

Chris blogs at [Wait till I come](#) and is available on many a social network as “codepo8”.

14: Choosing the right doctype for your HTML documents

BY ROGER JOHANSSON · 8 JUL, 2008

Published in: [DOCTYPE](#), [DTD](#), [HTML](#), [VALIDATION](#), [BROWSERS](#),

This is Article 14 of the Web Standards Curriculum.

[Previous article—The HTML <head> element](#)

[Next article—Marking up textual content in HTML](#)

[Table of contents](#)

Introduction

[Article 13](#) dissected the anatomy of the `head` section of an HTML document, looking briefly at what different things can be contained in the head, and what they do. In this article I will look at the doctype in a lot more detail, showing what it does and how it helps you validate your HTML, how to choose a doctype for your document, and the XML declaration, which you'll rarely need, but will sometimes come across.

- [The doctype comes first](#)
- [Doctype switching and rendering modes](#)
- [Validation](#)
- [Choosing a doctype](#)
- [The XML declaration](#)
- [Summary](#)
- [Exercise questions](#)
- [Further reading](#)

The doctype comes first

The very first thing you should make sure to have in any HTML document you create is a DTD declaration. If you haven't heard anyone mention a DTD declaration before, don't worry. For the sake of making things easier, it is often referred to as a "doctype", which is what I'll call it in the rest of this article.

You might be wondering what a "DTD" or doctype is. DTD is short for "Document Type Definition", and among other things it defines what elements and attributes are allowed to be used in a certain flavor of HTML—yes that's right, there are different versions of HTML in use on the Web today, but don't let this worry you—you'll only really need to concern yourselves with one.

The doctype is used for two things, by different kinds of software:

1. Web browsers use it to determine which rendering mode they should use (more on rendering modes later).
2. Markup validators look at the doctype to determine which rules they should check the document against (more on that later as well).

Both of these will affect you, but in different ways, which will be explained later on in this article.

Here is an example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
```

Now, that may look like a lot of nonsense to you, so let me offer a somewhat simplified explanation of how it is constructed. For a much more detailed look at exactly what each character refers to, see the article [!DOCTYPE](#).

The most important parts of the doctype are the two strings delimited by quotes. "`!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN`" states that this is a DTD document published by the W3C, that the DTD describes HTML version 4.01, and that the language used in the DTD is English.

The second string, "`http://www.w3.org/TR/html4/strict.dtd`", is a URL that points to the DTD document used for this doctype.

Even though a doctype may look a bit strange, it is required by the HTML and XHTML specifications. If you don't include one you will get a validation error when you check the syntax of your document with the W3C Markup validator or other tools that check HTML documents for errors. Some web browsers even contain such functionality by default, while others can have it added by installing an extension.

Doctype switching and rendering modes

If you do not provide a doctype, browsers will handle and render the document anyway—they need to make an attempt to render all sorts of strange things that they come across on the Web, so they can't always be very picky. However, without a doctype, the results may not look like you intended, because of something called "doctype sniffing" or "doctype switching".

Most web browsers released in the 21st century look at the doctype of any HTML documents they encounter and use that to decide whether the author of the documents took care to write their HTML and CSS properly according to web standards.

If they find a doctype that indicates that the document is coded well, they use something called "Standards mode" when they layout the page. In standards mode, browsers generally try to render the page according to the CSS specifications—they trust that the person who created the document knew what they were doing.

On the other hand, if they find an outdated or incomplete doctype, they use "Quirks mode", which is more backwards compatible with old practices and old browsers. Quirks mode assumes that the document is old or that it has not been created with web standards in mind—it means that the web page will still render, but it will take a lot more processing power to do so, and you'll likely get a strange or ugly result, which you weren't quite expecting.

The differences are mostly related to how CSS is rendered, and only in a few cases about how the actual HTML is treated. As a web designer or developer, you will get the most consistent results by making sure that all browsers use their Standards rendering mode, hence you should stick to web standards, and use a proper doctype!

Validation

As I mentioned earlier, the doctype is also used by validators, which you will learn more about later in this article series. For now, all you need to know is that a validator is used to check that the syntax of your HTML document is correct and does not contain any mistakes. The validator programs look at the doctype you have used to determine what rules to use. It's a bit like telling a spell checker which language a document is written in. If you don't tell it, it won't know which spelling and grammar rules to use.

Choosing a doctype

So, now that you know that you need to insert a doctype and what it is used for, how will you know which one to choose? There isn't just one, after all, there are many. You could even create your own if you feel up to something more advanced. But I'm not going to mention a whole lot of different doctypes. I'll try to keep things simple and settle for two.

If your document is HTML, use this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
```

```
"http://www.w3.org/TR/html4/strict.dtd">
```

If your document is XHTML, use this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
```

```
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Note: "Real" XHTML should be delivered to the web browser as XML, but the details of how and when to do that, and the implications it has, is beyond the scope of this particular article.

Both of these doctypes will ensure that browsers use their Standards mode when dealing with your document. The most noticeable effect that will have on your work is that you will get more consistent results when styling the document with CSS. To see some of the other doctypes that you could use, the W3C have published a list of [Recommended DTDs to use in your Web document](#).

You may notice that both of the doctypes I mention here are called "Strict". While that may sound a bit scary, it isn't.

There are strict and transitional flavours of both HTML and XHTML. Strict in this case means that the doctypes allow less presentational markup than the transitional doctype does. The presentational markup that isn't allowed shouldn't really be there anyway, since you should use HTML to define the structure and meaning of your documents, and CSS to determine how they are presented. Using a strict doctype will help you with that, since the validator will alert you of any presentational elements or attributes that have sneaked its way into your code.

The XML declaration

I stated earlier that the doctype needs to be the very first thing in your HTML documents. Well, that is in fact a slightly simplified version of the truth. There is also the XML declaration to consider.

You may have seen a code snippet that looks like this before the doctype in some XHTML documents:

```
<?xml version="1.0" encoding="UTF-8"?>
```

This is called an XML Declaration, and when it is present it needs to be inserted *before* the doctype.

Internet Explorer version 6 has a problem with that—this causes it to switch into Quirks mode, and as I explained earlier you most likely do not want that.

Luckily the XML declaration is not required unless you are really sending your XHTML documents as XML to web browsers (see the sidenote about XHTML) *and* you are using a different character encoding than UTF-8 *and* your server is not sending an HTTP header that determines the character encoding.

The probability of all that happening all at once is quite slim, so the easiest way to solve the Internet Explorer problem is to simply omit the XML declaration. Don't forget the doctype though!

Summary

Always include one of the doctypes mentioned here as the very first thing in all of your HTML documents. It will make sure that validators know what version of HTML you are using, so they can correctly report any mistakes you have made. It will also make sure that all recent web browsers use their Standards mode, which will give you more consistent results when you are styling the document with CSS.

Exercise questions

Here a few questions that you should be able to answer after reading this:

- What are the two main purposes of including a doctype in HTML documents?
- What are the benefits of using a strict doctype instead of a transitional one?
- Why is the XML declaration problematic?
- One doctype I haven't mentioned in this article is the frameset doctype—research what this does, and why it shouldn't be used.

Further reading

- [Don't forget to add a doctype](#)
- [Recommended DTDs to use in your Web document.](#)
- [Fix Your Site With the Right DOCTYPE!](#)
- [Activating the Right Layout Mode Using the Doctype Declaration](#)
- [The Opera 9 DOCTYPE Switches](#)
- [Quirks mode and strict mode](#)
- [Transitional vs. Strict Markup](#)

About the author



Roger Johansson is a web professional with a passion for web standards, accessibility, and usability. He spends his days developing websites at Swedish web consultancy [NetRelations](#), and his evenings and weekends writing articles for his personal sites [456 Berea Street](#) and [Kaffesnobben](#).

When he is not stuck in front of a computer, Roger can often be found either in his garden, getting dirt under his fingernails, or in the wilderness, fishing.

15: Marking up textual content in HTML

BY [MNFRANCIS](#) · 8 JUL, 2008

Introduction

In this article I will take you through the basics of using HTML to describe the meaning of the content within the body of your document.

We will look at general structural elements such as headings and paragraphs and embedding quotes and code. After that we will look at inline content, such as short quotes and emphasis, and finish with a quick examination of old-fashioned presentational content. This article's structure is as follows:

- [Space—the final frontier](#)
- [Block level elements](#)
 - [Page section headings](#)
 - [Generic paragraphs](#)
 - [Quoting other sources](#)
 - [Preformatted text](#)
- [Inline elements](#)
 - [Short quotations](#)
 - [Emphasis](#)
 - [Italicised text](#)
- [Presentational elements—never use these](#)
- [Summary](#)

Note: After each code example, there is a “View source” link, which when clicked will take you to the actual rendered output of that source code, contained within a different file—it is to view live examples of how the source code is actually rendered in the browser, as well as looking at the code.

Space—the final frontier

An important point to cover before I start discussing text, though, is that of space, specifically the space between words. When writing HTML, the source document will contain what is termed “white space” — the characters in the file that serve to separate text. An actual space character, as you would get when you hit the Spacebar on the keyboard is the most common, but there are others such as the Tab character and the marker between two separate lines in a document (called a carriage return or new line).

In HTML, multiple occurrences of these characters are (almost) always treated as a single space character. For example:

```
<h3>In    the  
beginning</h3>
```

[View live examples](#)

would be interpreted by a web browser to be equivalent to:

```
<h1>In the beginning</h1>
```

The only place where this is not the case is in the `pre` element, which is discussed in detail later in this article.

This can be a source of confusion for first-time authors of an HTML document, who try to pad out text with extra spaces to achieve a desired indentation, or to get more spacing after the period between sentences and introduce more vertical space between paragraphs. Influencing the visual layout of your documents is not something to be done in the HTML source, and is instead achieved through style sheets, discussed later in this series of articles.

Block level elements

In this section I'll go through the syntax and usage of the common [block level elements](#) used to format text.

Page section headings

Once the page has been broken down into logical sections, each section should be introduced by an appropriate header. This is discussed further in the article [What does a good web page need](#).

HTML defines six levels of header, h1, h2, h3, h4, h5, and h6 (from the highest importance to the lowest). Generally speaking, the h1 would be the main heading of the entire page and introduce everything. h2 is then used to break the page up into sections, h3 the sub-sections, and so on.

It is important to use the header levels to describe the document in terms of section, sub-section, sub-sub-section as this makes the [document more understandable to screen readers](#) and to automated processes (like Google's indexing bots).

A good example of a header structure, using this document as a template, would look like this:

```
<h1>Marking up Textual Content, the Basics</h1>
<h2>Introduction</h2>
<h2>Space, the final frontier</h2>
<h2>Block level elements</h2>
  <h3>Page section headings</h3>
  <h3>Generic paragraphs of text</h3>
  <h3>Quoting other documents or sources</h3>
  <h3>Preformatted text</h3>
<h2>Inline elements</h2>

[...and so on...]
```

[View live examples](#)

Generic paragraphs

The paragraph is the building block of most documents. In HTML a paragraph is represented by the p element, which takes no special attributes. For example:

```
<p>This is a very short paragraph. It only has two sentences.</p>
```

[View live examples](#)

A paragraph in many articles and books can contain just one sentence. Whilst the meaning (in terms of written prose) of "paragraph" is fairly clear, on the web much shorter areas of text are often wrapped in paragraph elements as the author believes this is more "semantic" than using a div element (we will cover these in a future article called "Generic containers").

A paragraph is a collection of one or more sentences, just as in newspapers and books. On the web, it is good form to use the paragraph element for this and not just any random piece of text in the page. If it is just a few words and not even a proper sentence, then it should probably not be marked up as a paragraph.

Quoting other sources

Very often articles, blog posts, and reference documents will quote in whole or in part another document. In HTML, this is marked up using the `blockquote` element for lengthy quotations, such as entire sentences, paragraphs, lists, or the like.

A `blockquote` element cannot contain text, but must instead have another block level element inside it. You should use the same block level element as was used in the original document. If you are quoting a paragraph of text, use a paragraph; when quoting a list of items, use the elements for lists; and so on.

If the quote comes from another web page, you can indicate this using the `cite` attribute, like so:

```
<p>HTML 4.01 is the only version of HTML that you should use
when creating a new web page, as, according to the
specification:</p>
<blockquote cite="http://www.w3.org/TR/html401/">
<p>This document obsoletes previous versions of HTML 4.0,
although W3C will continue to make those specifications and
their DTDs available at the W3C Web site.</p>
</blockquote>
```

[View live examples](#)

The `cite` attribute is not required in the case where the quote is taken from a novel, magazine or other form of offline content.

Preformatted text

Any text in which the formatting and white space (see earlier) is significant should be marked up using the `pre` element.

In most web browsers, text marked as preformatted will be displayed to the user as it appears in the source, sometimes using a fixed-width (monospaced) font, giving the text a feeling of having come from a typewriter. This is an artifact of programmers using fixed width fonts for early uses of preformatted text.

In this example, you can see a snippet of code written in the perl programming language:

```
<pre><code class="language-perl">
# read in the named file in its entirety
sub slurp {
    my $filename = shift;
    my $file      = new FileHandle $filename;

    if ( defined $file ) {
        local $/;
        return <$file>;
    }
    return undef;
};
</code></pre>
```

[View live examples](#)

The use of `code` above will be explained in the [lesser-known semantic elements](#) article later on in the course.

Inline elements

In this section I'll go through the syntax and usage of the common [inline elements](#) used to format text.

Short quotations

Short quotes which are used within a normal sentence or paragraph are contained within the `q` element. Like the `blockquote` element, this can contain a `cite` attribute, which indicates the page on the internet where the quote can be found.

A short quote should normally be rendered with quotation marks around it. According to [the HTML specification](#), these should be inserted by the user-agent so that they can be correctly nested and made aware of the language being used in the document. CSS can be used to control the quotation marks used—this is covered in a later article on “styling text”.

An example of `q` in action:

```
<p>This did not end well for me. Oh well,  
    <q lang="fr">c'est la vie</q> as the French say.</p>
```

[View live examples](#)

Emphasis

HTML contains two methods for indicating that the text within needs to be emphasised to the user, such as error messages, warnings, or notes. For visual browsers this normally means applying a different colour, font or making the text bolder or italicised. For users of screen readers this can result in a different voice or other auditory effect.

For text that needs to be emphasised, you use the `em` element, like so:

```
<p><em>Please note:</em> the kettle is to be unplugged at  
    night.</p>
```

[View live examples](#)

If an entire sentence was to be emphasised, but there was still a point within that sentence needed to be emphasised further, you use the `strong` element to indicate stronger emphasis than normal, like so:

```
<p><em>Please note: the kettle <strong>must</strong> be unplugged every evening,  
otherwise it will explode -  
<strong>killing us all</strong></em>.</p>
```

[View live examples](#)

Italicised text

It is commonly thought that “italicised” does not describe the meaning, and thus the `i` element should not be used (much like some other presentational elements described in the next section).

There are a couple of instances when describing the content as being italicised is arguably correct. It has been noted that some concepts are best described as “italicised” rather than having to create some very specific and otherwise unused elements. These include things such as the names of ships, the titles of television series, movies and books, some technical terms and other taxonomic designations.

The argument is that the italicisation indicates that the text within is special, and the context indicates how it is special. Indeed, this is reflected in the currently draft HTML 5 specification:

The `i` element represents a span of text in an alternate voice or mood, or otherwise offset from the normal prose [...] The `i` element should be used as a last resort when no other element is more appropriate.

Since the `i` element can be restyled by CSS to not be italic, the meaning of “italic” in this context is essentially “something a little bit different”. I don’t find this acceptable, personally speaking, but there is enough precedent out there for it to be used this way.

Presentational elements—never use these

The HTML specification includes several elements that are widely described as “presentational” because they only specify what the content within them should look like, and not what it means.

Some of these have been labeled as deprecated in the specification. This means that they have been superseded by a newer method of achieving the same result.

I will describe them briefly here, but note that this is mostly of historic interest—these elements should never be used in any modern web page. The effect of all of these elements should be achieved in another way and will be described in two forthcoming articles: “styling text with CSS” and [lesser known semantic elements](#).

`font face="..." size="..."`

The text within should be rendered by the browser using a font different from the default — instead, fonts should be set using CSS.

`b`

The text within is bold—this almost always means the text has been emphasised, so you should use `em` or `strong` as shown earlier.

`s` and `strike`

The text within has been struck-through with a line—if this is merely a presentational effect, this should be achieved with CSS. Alternatively, if the text is actually being marked as having been deleted or unwanted it should be marked up with the `del` element, described in the later article.

`u`

The text within has been underlined—this is almost always a visual effect, and so should be achieved with CSS.

`tt`

The text within is presented in a “teletype” or monospaced font —this should be achieved with CSS or a more appropriate semantic element such as `pre`—as shown above.

`big` and `small`

The size of the text within has been adjusted—this should be achieved with CSS.

Summary

In this article, I have talked about some of the most common elements used when marking up textual content. [In the next article](#), you will progress to another type of content: lists of items.

About the author



Photo credit: [Andy Budd](#).

Mark Norman Francis has been working with the internet since before the web was invented. He currently works at Yahoo! as a Front End Architect for the world’s biggest website, defining best practices, coding standards and quality in web development internationally.

Previous to Yahoo! he worked at Formula One Management, Purple Interactive and City University in various roles including web development, backend CGI and systems architecture. He blogs at <http://marknormanfrancis.com/>.

16: HTML lists

BY [BEN BUCHANAN](#) · 8 JUL, 2008

Introduction

Lists are used to group related pieces of information together, so they are clearly associated with each other and easy to read. In modern web development lists are workhorse elements, frequently used for navigation as well as general content.

Lists are good from a structural point of view as they help create a well-structured, more accessible, easy-to-maintain document. They are also useful for a purely practical reason—they give you extra elements to attach CSS styles to, for a whole variety of styling.

In this article, I will cover the different list types available in HTML, when and how you should use them, and how to apply some basic styles. The table of contents is as follows (oooh look—a list!):

- [The three list types](#)
 - [Unordered lists](#)
 - [Ordered lists](#)
 - [Ordered list markup](#)
 - [Beginning ordered lists with numbers other than 1](#)
 - [Definition lists](#)
- [Choosing between list types](#)
- [The difference between HTML lists and text](#)
- [Nesting lists](#)
- [Step by step example](#)
 - [Main page markup](#)
 - [Adding some style](#)
 - [The recipe page](#)
 - [Recipe page markup](#)
 - [Styling the recipe page](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

The three list types

There are three list types in HTML:

- unordered list—used to group a set of related items, in no particular order.
- ordered list—used to group a set of related items, in a specific order.
- definition list—used to display name/value pairs such as terms and their definitions, or times and events.

Each one has a specific purpose and meaning—they are not interchangeable!

Unordered lists

Unordered lists, or bulleted lists, are used when a set of items can be placed in any order. An example is a shopping list:

- milk
- bread
- butter
- coffee beans

These items are all part of one list, however, you could put the items in any order and the list would still make sense:

- bread
- coffee beans
- milk
- butter

You can use CSS to change the bullet to one of several default styles, use your own image, or even display the list without bullets—we'll look at how to do that a bit later in this article, and expand on it a lot more in a future article.

Unordered list markup

Unordered lists use one set of `` tags, wrapped around many sets of ``:

```
<ul>
  <li>bread</li>
  <li>coffee beans</li>
  <li>milk</li>
  <li>butter</li>
</ul>
```

Ordered lists

Ordered lists, or numbered lists, are used to display a list of items that need to be placed in a specific order. An example would be cooking instructions, which must be completed in order for the recipe to work:

1. Gather ingredients
2. Mix ingredients together
3. Place ingredients in a baking dish
4. Bake in oven for an hour
5. Remove from oven
6. Allow to stand for ten minutes
7. Serve

If the list items were moved around into a different order, the information would no longer make sense:

1. Gather ingredients
2. Bake in oven for an hour
3. Remove from oven
4. Serve
5. Place ingredients in a baking dish
6. Allow to stand for ten minutes
7. Mix ingredients together

Ordered lists can be displayed with one of several numbering or alphabetic systems—that is, letters or numbers. The default in most browsers is decimal numbers, but there are more options:

- Letters
 - Lowercase ascii letters (a, b, c...)
 - Uppercase ascii letters (A, B, C...).
 - Lowercase classical Greek: (α, β, γ...)
- Numbers
 - Decimal numbers (1, 2, 3...)
 - Decimal numbers with leading zeros (01, 02, 03...)
 - Lowercase Roman numerals (i, ii, iii...)
 - Uppercase Roman numerals (I, II, III...)
 - Traditional Georgian numbering (an, ban, gan...)
 - Traditional Armenian numbering (mek, yerku, yerek...)

Again, you can use CSS to change the style of your lists.

Ordered list markup

Ordered lists use one set of `` tags, wrapped around many sets of ``:

```
<ol>
  <li>Gather ingredients</li>
  <li>Mix ingredients together</li>
  <li>Place ingredients in a baking dish</li>
  <li>Bake in oven for an hour</li>
  <li>Remove from oven</li>
  <li>Allow to stand for ten minutes</li>
  <li>Serve</li>
</ol>
```

Beginning ordered lists with numbers other than 1

It is possible to get an ordered list to start with a number other than 1 (or i, or I, etc.). This is done using the start attribute, which takes a numeric value, even if you're using CSS to change the the list counters to be alphabetic or roman using the `list-style-type` property. This is useful if you have a single list of items, but you want to break the list up with some kind of note, or some other related information. For example, we could do this with the previous example:

```
<ol>
  <li>Gather ingredients</li>
  <li>Mix ingredients together</li>
  <li>Place ingredients in a baking dish</li>
</ol>

<p class="note">Before you place the ingredients in the baking dish, preheat the
oven to 180 degrees centigrade/350 degress farenheit in readiness for the next
step</p>

<ol start="4">
  <li>Bake in oven for an hour</li>
  <li>Remove from oven</li>
  <li>Allow to stand for ten minutes</li>
  <li>Serve</li>
</ol>
```

This gives the following result:

1. Gather ingredients
2. Mix ingredients together
3. Place ingredients in a baking dish

Before you place the ingredients in the baking dish, preheat the oven to 180 degrees centigrade/350 degress farenheit in readiness for the next step

4. Bake in oven for an hour
5. Remove from oven
6. Allow to stand for ten minutes
7. Serve

Note that this attribute is actually deprecated in the latest version of the HTML spec, which means that it will cause your pages to not validate when using strict doctypes. This may seem odd, as the attribute makes sense, and there is no CSS equivalent. This shows that validating HTML is an ideal goal to follow, but not always the absolute be all and end all. In addition, we have another leg to stand on - the start attribute is no longer deprecated in the HTML5 spec (the [HTML 5 differences from HTML4](#) document attests

to this). If you want to make use of such functionality in an HTML4 strict page, and it absolutely has to validate, you can do it using CSS Counters instead.

Definition lists

Definition lists associate specific items and their definitions within the list. For example, if you wanted to give a definition to the items on your shopping list, you could do that like so:

milk

A white, liquid dairy product.

bread

A baked food made of flour or meal.

butter

A yellow, solid dairy product.

coffee beans

The seeds of the fruit from certain coffee trees.

Each definition and term is a definition group (or name-value group). You can have as many definition groups as you like, but there must be at least one term and at least one definition in each group. You can't have a term with no definition or a definition with no term.

You can associate more than one term with a single definition, or vice versa. For example, the term "coffee" can have several meanings, and you could show them one after the other:

coffee

a beverage made from roasted, ground coffee beans

a cup of coffee

a social gathering at which coffee is consumed

a medium to dark brown colour

Alternatively you can have more than one term with the same definition. This is useful to show variations of a term, all of which have the same meaning:

soda

pop

fizzy drink

cola

A sweet, carbonated beverage.

Definition lists are different from the other kinds of list, as they use definition terms and definition descriptions instead of list items.

So, definition lists use one set of `<dl></dl>` elements, wrapped around groups of `<dt></dt>` and `<dd></dd>` tags. You must pair at least one `<dt></dt>` with at least one `<dd></dd>`; the `<dt></dt>` should always be first in the source order.

A simple definition list of single terms with single definitions would look like this:

```
<dl>
  <dt>Term</dt>
  <dd>Definition of the term</dd>
  <dt>Term</dt>
  <dd>Definition of the term</dd>
  <dt>Term</dt>
  <dd>Definition of the term</dd>
</dl>
```

Which are rendered as follows:

Term

Definition of the term

Term

Definition of the term

Term

Definition of the term

In this example, we associate more than one term with a definition, and vice versa:

```
<dl>
  <dt>Term</dt>
  <dd>Definition of the term</dd>
  <dt>Term</dt>
  <dt>Term</dt>
  <dd>Definition that applies to both of the preceding terms</dd>
  <dt>Term that can have both of the following definitions</dt>
  <dd>One definition of the term</dd>
  <dd>Another definition of the term</dd>
</dl>
```

Which would render as follows:

Term

Definition of the term

Term

Term

Definition that applies to both of the preceding terms

Term that can have both of the following definitions

One definition of the term

Another definition of the term

In general, it is unusual to associate multiple terms with a single definition, but it is useful to know it is possible in case the need arises.

Choosing between list types

When trying to decide what type of list to use, you can usually decide by asking two simple questions:

1. Am I defining terms (or associating other name/value pairs)?
 - If yes, use a definition list.
 - If no, don't use a definition list—go on to the next question.
2. Is the order of the list items important?
 - If yes, use an ordered list.
 - If no, use an unordered list.

The difference between HTML lists and text

You may be wondering what the difference is between an HTML list and some text with bullets or numbers written in by hand. Well, there are several advantages to using an HTML list:

- If you have to change the order of the list items in an ordered list, you simply move them around in the HTML. If you wrote the numbers in manually you would have to go through and change every single item's number to correct the order—which is tedious to say the least!
- Using an HTML list allows you to style the list properly. If you just use a blob of text, you will find it more difficult to style the individual items in any useful manner.
- Using an HTML list gives the content the proper semantic structure, rather than just a "list-ish" visual effect. This has important benefits such as allowing screen readers to tell users with visual impairments they are reading a list, rather than just reading out a confusing jumble of text and numbers.

To put it another way: text and lists are not the same. Using text instead of a list makes more work for you and can create problems for your document's readers. So if your document needs a list, you should use the correct HTML list.

Nesting lists

A list item can contain another entire list—this is known as "nesting" a list. It is useful for things like tables of contents, such as the one at the start of this article:

1. Chapter One
 1. Section One
 2. Section Two
 3. Section Three
2. Chapter Two
3. Chapter Three

The key to nesting lists is to remember that the nested list should relate to one specific list item. To reflect that in the code, the nested list is contained inside that list item. The code for the list above looks as follows:

```
<ol>
  <li>Chapter One
    <ol>
      <li>Section One</li>
      <li>Section Two </li>
      <li>Section Three </li>
    </ol>
  </li>
  <li>Chapter Two</li>
  <li>Chapter Three </li>
```

```
</ol>
```

Note how the nested list starts after the `` and the text of the containing list item (“Chapter One”); then ends before the `` of the containing list item. Nested lists often form the basis for website navigation menus, as they are a good way to define the structure of the website.

Theoretically you can nest as many lists as you like, although in practice it can become confusing to nest lists too deeply. For very large lists, you may be better off splitting the content up into several lists with headings instead, or even splitting it up into separate pages.

Step by step example

Let’s run through a step by step example, to put all of this together. Consider the following scenario:

You are creating a small website for the HTML Cooking School. On the main page, you are to show a list of categorised recipes, linking through to recipe pages. Each recipe page lists the ingredients required, notes on those ingredients and the preparation method. The three categories are Cakes (including recipes for “Plain Sponge”, “Chocolate Cake” and “Apple Tea Cake”); “Biscuits” (including recipes for “ANZAC Biscuits”, “Jam Drops” and “Melting Moments”); and “Quickbreads” (including recipes for “Damper” and “Scones”). The client doesn’t mind what order the categories and recipes are shown; they just want to be sure people know which items are categories and which ones are recipes.

Let’s step through the process of creating this site. In this section I’ll take you through building up the markup, and also show you how to add some styling to your lists. The styling won’t be explained in much detail until the article on styling lists later on in the series.

Main page markup

1. Create a properly-formed HTML page, including doctype, `html`, `head` and `body` elements, and save it as *stepbystep-main.html*. Add a main heading (`h1`) of “HTML Cooking School”, and a subheading (`h2`) of “Recipes”:

```
2.      <h1>HTML Cooking School</h1>
```

```
3.
```

```
      <h2>Recipes</h2>
```

4. You have three categories of recipe to represent, and the order is not important—an unordered list is most appropriate for these, so add the following to your page:

```
5.      <h2>Recipes</h2>
6.      <ul>
7.      <li>Cakes</li>
8.      <li>Biscuits</li>
9.      <li>Quickbreads</li>
```

```
</ul>
```

Indenting the `li` elements makes the code more readable, but it is not required.

10. Now you need to add the recipes as sub-items, for example “Plain Sponge”, “Chocolate Cake” and “Apple Tea Cake” are all part of the “Cakes” category. To do this, you need to create a nested list within each item. Since the order is not important, once again unordered lists are appropriate. To make things easier for the tutorial, I’ll get you to link all of the recipes to one single recipe page:

```
11.      <h2>Recipes</h2>
12.      <ul>
```

```

13.     <li>Cakes
14.     <ul>
15.     <li><a href="stepbystep-recipe.html">Plain Sponge</a></li>
16.     <li><a href="stepbystep-recipe.html">Chocolate Cake</a></li>
17.     <li><a href="stepbystep-recipe.html">Apple Tea Cake</a></li>
18.     </ul>
19.     </li>
20.     <li>Biscuits
21.     <ul>
22.     <li><a href="stepbystep-recipe.html">ANZAC Biscuits</a></li>
23.     <li><a href="stepbystep-recipe.html">Jam Drops</a></li>
24.     <li><a href="stepbystep-recipe.html">Melting Moments</a>      </li>
25.     </ul>
26.     </li>
27.     <li>Breads/quickbreads
28.     <ul>
29.     <li><a href="stepbystep-recipe.html">Damper</a></li>
30.     <li><a href="stepbystep-recipe.html">Scones</a></li>
31.     </ul>
32.     </li>

```

```
</ul>
```

Adding some style

The client likes this arrangement, but wants the categories to have small arrows instead of bullets. They also want the categories to sit flush left on the page. To achieve this you need to specify an image instead of a bullet, then adjust the margin/padding settings.

1. To avoid clashing with other lists on the site, you should add a class to the containing list, so you can make specific contextual selectors in your stylesheet. The class "recipe-list" seems appropriate:

```
2.     <h2>Recipes</h2>
```

```
3.
```

```
<ul class="recipe-list">
```

4. Now you need to create a stylesheet, and add some rules to it—first add opening and closing style tags into the head of your document.
5. Now you will remove the spacing from the containing list. By default some browsers use margin and some use padding to space elements, so you need to set both to zero—add the following between your style tags:

```

6.     ul.recipe-list {
7.         margin-left: 0;
8.         padding-left: 0;

```

```

    }

```

9. Next, create a custom bullet image—you can use mine if you like (see Figure 1).



Figure 1: Custom bullet image.

10. Now you will remove the bullets from the list items and set the bullet as the background image for the list items, adding some padding so the text doesn't sit on top of the background image. You can do this by adding the following CSS just before the closing style tag:

```
11.    ul.recipe-list li {  
12.        list-style-type: none;  
13.        background: #fff url("example-bullet.gif") 0 0.4em no-repeat;  
14.        padding-left: 10px;  
  
    }
```

15. Finally, you will put the bullets back onto the nested list items and set the background to plain white (the second rule is more specific, so it will override the background image rule). Remember that the first CSS rule will be inherited by the nested list, so you need to "undo" all container settings. Add the following CSS just before the closing `style` tag:

```
16.    ul.recipe-list li li {  
17.        list-style-type: disc;  
18.        background: #fff;  
19.    }
```

The final result should be something similar to Figure 2:

HTML Cooking School

Recipes

> Cakes

- [Plain Sponge](#)
- [Chocolate Cake](#)
- [Apple Tea Cake](#)

> Biscuits

- [ANZAC Biscuits](#)
- [Jam Drops](#)
- [Melting Moments](#)

> Breads/quickbreads

- [Damper](#)
- [Scones](#)

Figure 2: The finished main page, with custom bullet images.

You can also [view the live example page here](#).

The recipe page

For the sake of the example, we will just create the sponge cake recipe page—feel free to create the others yourself, using this one as a template. The client has supplied the sponge recipe to you in a text file, looking like this:

```
Simple Sponge Cake  
Ingredients  
3 eggs  
100g castor sugar  
85g self-raising flour  
Notes on ingredients:  
    Caster Sugar - Finely granulated white sugar.
```


Self-raising flour - A pre-mixed combination of flour and leavening agents (usually salt and baking powder).

Method

1. Preheat the oven to 190°C.
2. Grease a 20cm round cake pan.
3. In a medium bowl, whip together the eggs and castor sugar until fluffy.
4. Fold in flour.
5. Pour mixture into the prepared pan.
6. Bake for 20 minutes in the preheated oven, or until the top of the cake springs back when lightly pressed.
7. Cool in the pan over a wire rack.

Recipe page markup

1. Create another properly-formed HTML document, and save it as `stepbystep-recipe.html`. Add the following headings to it:

2. `<h1>Simple Sponge Cake</h1>`
3. `<h2>Ingredients</h2>`
4. `<h3>Notes on ingredients</h3>`

`<h2>Method</h2>`

5. The ingredients list has several items but the order isn't important. An unordered list therefore makes sense. Add the following into the `body` of your HTML:

6. `<h2>Ingredients</h2>`
7. ``
8. `3 eggs`
9. `100g castor sugar`
10. `85g self-raising flour`

``

11. The notes on the ingredients are there to properly define what some of the ingredients are. You need to associate the ingredient—the term—with its definition. A definition list is right for this purpose. Add the following to your HTML, below the unordered list in the previous step:

12. `<h3>Notes on ingredients</h3>`
13. `<dl>`
14. `<dt>Castor Sugar</dt>`
15. `<dd>Finely granulated white sugar.</dd>`
16. `<dt>Self-raising flour</dt>`
17. `<dd>A pre-mixed combination of flour and leavening agents (usually salt and baking powder).</dd>`

`</dl>`

18. The method must obviously follow a single correct order, so it should be an ordered list—add the following to your HTML, below the definition list:

19. `<h2>Method</h2>`
20. ``
21. `Preheat the oven to 190°C.`
22. `Grease a 20cm round cake pan.`
23. `In a medium bowl, whip together the eggs and castor sugar until fluffy.`
24. `Fold in flour.`
25. `Pour mixture into the prepared pan.`
26. `Bake for 20 minutes in the preheated oven, or until the top of the cake springs back when lightly pressed.`

```
27.      <li>Cool in the pan over a wire rack.</li>
</ol>
```

Styling the recipe page

The client is happy with this but wants the definitions to be bold, to aid readability. Add the following inside your HTML head:

```
<style>
dt {
  font-weight: bold;
}
</style>
```

The page should look something like Figure 3:

Simple Sponge Cake

Ingredients

- 3 eggs
- 100g castor sugar
- 85g self-raising flour

Notes on ingredients

Castor Sugar

Finely granulated white sugar.

Self-raising flour

A pre-mixed combination of flour and leavening agents (usually salt and baking powder).

Method

1. Preheat the oven to 190°C.
2. Grease a 20cm round cake pan.
3. In a medium bowl, whip together the eggs and castor sugar until fluffy.
4. Fold in flour.
5. Pour mixture into the prepared pan.
6. Bake for 20 minutes in the preheated oven, or until the top of the cake springs back when lightly pressed.
7. Cool in the pan over a wire rack.

Figure 3: The finished recipe page with the definition terms in bold.

You can also [view the live example page here](#).

You're done!

Summary

By this stage you should have a clear understanding of the three different list types in HTML. Using the step-by-step example, you should have created all three and learned how to nest lists inside list items.

Once you know how to use HTML lists properly, you will probably discover that you use them all the time. There is a lot of content on the web that should have been placed into a list, but was just thrown into a generic element with some line break tags. It's a lazy practice which causes far more problems than it solves—so don't do it! You should always create semantically correct lists to help people read your websites. It is a better practice for everyone, not least yourself when you need to maintain your sites later on.

Further reading

- [A List Apart: Taming Lists](#)
- [W3C CSS2: list-style-type definition](#)

Exercise questions

Questions you should be able to answer:

- What are the three types of HTML list?
- When would you use each type of list? How would you choose between them?
- How do you nest lists?
- Why should you use CSS rather than HTML to style your lists?

About the author



Ben Buchanan started creating web pages more than ten years ago, while completing a degree in everything but IT. He has worked in both the public (university) and private sectors; and worked on the redevelopment of major websites including [The Australian](#) and three generations of [Griffith University](#)'s corporate website. He now works as Frontend Architect for [News Digital Media](#) and writes at [the 200ok weblog](#).

17: Images in HTML

BY [CHRISTIAN HEILMANN](#) · 8 JUL, 2008

Introduction

In this tutorial I'll talk about one of the things that makes web design pretty—images. At the end of this tutorial you'll know how to add imagery to web documents in an accessible way (so that people with visual impairments can still use the information on your site) and how and when to use inline images for delivering information or background images for page layout. You can [download the example files used in this article here](#)—I'll refer to these files during the course of the tutorial. The contents are as follows:

- [A picture says more than a thousand words—or does it?](#)
- [Different types of images on the web—content and background images](#)
- [The `img` element and its attributes](#)
 - [Providing a text alternative with the `alt` attribute](#)
 - [Adding nice-to-have information using the `title` attribute](#)
 - [Using `longdesc` to provide an alternative for complex images](#)
 - [Faster image display by defining the dimensions using `width` and `height`](#)
- [So much for inline images](#)
- [Background images with CSS](#)
 - [How to apply backgrounds with CSS](#)
- [Summary](#)
- [Exercise questions](#)

A picture says more than a thousand words—or does it?

It is very tempting to use a lot of imagery on your web sites. Images are a great way to set the mood for the visitor and illustrations are a nice way to make complex information easier to take in for visual learners.

The drawback of images on the web is that not everybody who surfs the web can see them. Back in the days when images were first supported by browsers, many site visitors had images turned off, to save on traffic and get a faster surfing experience—web connections used to be very slow, and you'd pay a lot of money for each minute you were online. While this is not very common these days, we're still not out of the woods—by a long shot:

- People surfing on mobile devices might still have images turned off because of small screens and the cost of downloading data.
- Visitors of your site might be blind or visually impaired to such a degree that they cannot see your images properly.
- Other visitors might be from a different culture and not understand the icons you use.
- Search engines only index text—they don't analyze images (yet), which means that information stored in images cannot be found and indexed.

It is therefore very important to choose images wisely and only use them when appropriate. It is even more important to make sure you always offer a fallback for those who cannot see your images. There is more on the problems of wrongly used icons and images in the “Web navigation and menus” tutorial later on in the series (to be published soon). For now, let's see what technologies are available to add images to an HTML document.

Different types of images on the web—content and background images

There are two main ways to add images to a document: content images using the `img` element and background images applied to elements using CSS. When to use what is dependent on what you want to do:

1. If the image is crucial to the content of the document, for example a photo of the author or a graph showing some data, it should be added as an `img` element with proper alternative text.
2. If the image is there as “eye candy” you should use CSS background images. The reason is that these images should not have any alternative text (what use is “round green corner with a twinkle” to a blind person?) and you have a lot more options to deal with image styling in CSS than in HTML.

The `img` element and its attributes

Adding an image to an HTML document is very easy using the `img` element. The following HTML document ([inlineimageexample.html](#) in the zip file) displays the photo `balconyview.jpg` in a browser (provided that you have the image in the same folder as the HTML file.)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Example of an inline image</title>
</head>
<body>

</body>
</html>
```

If you run this code in a browser, you'll get an output as shown in Figure 1.

Figure 1: The image as it is shown in a browser.

Providing a text alternative with the `alt` attribute

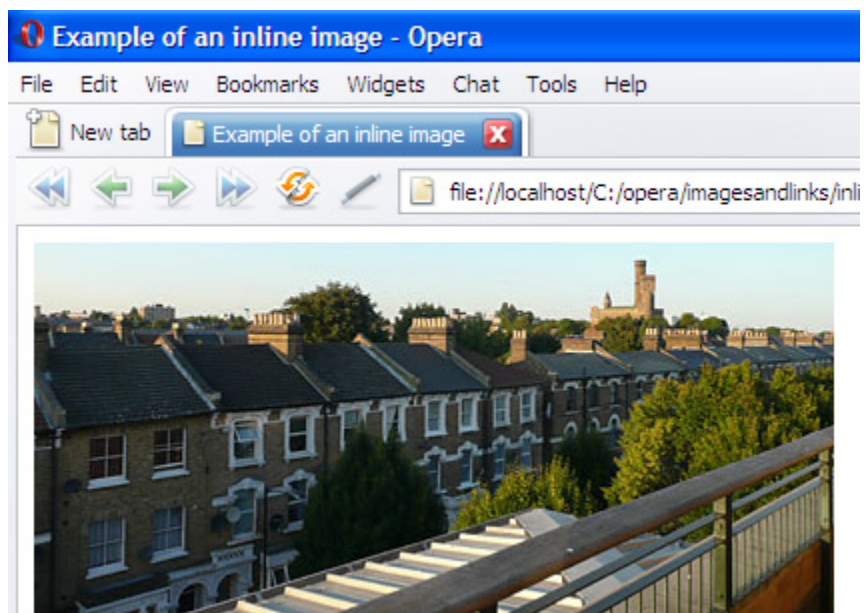
This displays the image fine, however, it is invalid HTML because the `img` element needs an `alt` attribute. This attribute contains text that is displayed if the image is not available for some reason. The image may not be available because it could not be found, loaded or because the user agent (normally a browser) does not support images. In

addition, people with visual disabilities use assistive technologies to read web pages to them. These technologies will read the contents of the `alt` attribute of `img` elements out to their users. It is therefore important to write good alternative text to describe the contents of the image and put it in the `alt` attribute.

You'll find a lot of texts on the web talking about “alt tags”. This is factually wrong as there is no tag (or element) with that name. It is an attribute of the `img` element and amazingly important both for accessibility and search engine optimization.

In order to make the image understandable for everybody you need to add a proper alternative text, for example in this case “View from my balcony, showing a row of houses, trees and a castle”

([inlineimageexamplealt.html](#)):



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Example of an inline image</title>
</head>
<body>

</body>
</html>
```

The `alt` attribute contains the text that should be displayed when the image is not available. The information in the `alt` attribute should not be displayed when the image was successfully loaded and shown; Internet Explorer gets this wrong, and shows it as a tooltip when you hover your mouse pointer over the image for a while. This is a mistake, as it leads a lot of people to add additional information about the image into the `alt` attribute. If you wanted to add additional information, you should use the `title` attribute instead, which I'll get on to in the next section.

Adding nice-to-have information using the title attribute

Most browsers will display the value of an `img` element's `title` attribute as a tool-tip when you hover your mouse cursor over it (see Figure 2.) This can help a visitor learn more about the image, but you cannot rely on each visitor to have a mouse. The `title` attribute can be very useful, but it is not a safe way of providing crucial information. Instead it offers a good way to, for example, write about the mood of the image, or what it means in context ([inlineimagewithtitle.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Example of an inline image with alternative text and title</title>
</head>
<body>

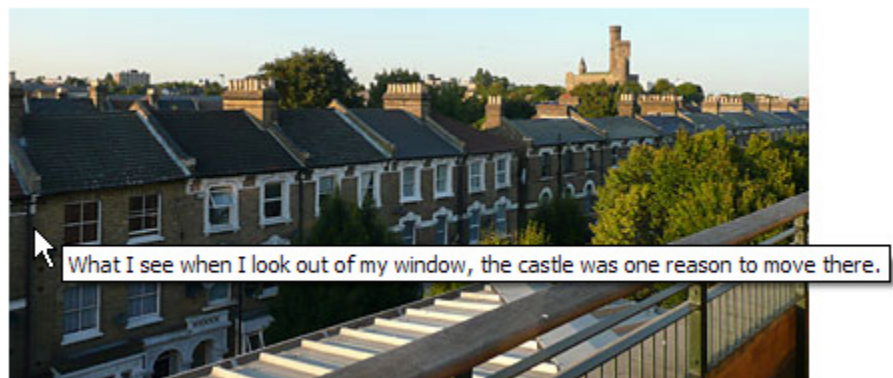
</body>
</html>
```

If you load this code in your browser, you will see the display shown in Figure 2.

Figure 2: `title` attributes are shown as tool tips in a lot of browsers.

Using `longdesc` to provide an alternative for complex images

If the image is a very complex image, like for example a chart, you can offer a more lengthy description of it using the `longdesc` attribute, so that people using screenreaders or browsing with images turned off can still access the information conveyed by the image.



This attribute contains a URL that points to a document containing the same information. For example, if you have a chart showing a set of data, you can link it to a data table with the same information using `longdesc` ([inlineimagelongdesc.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Example of an inline image with longdesc</title>
</head>
<body>

</body>
</html>
```

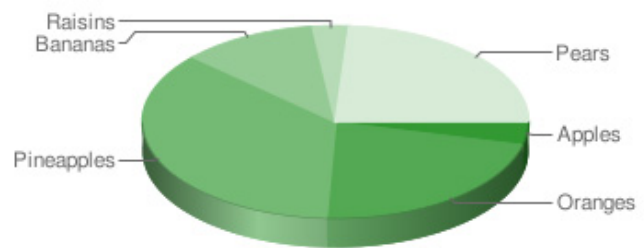
The data file [fruitconsumption.html](#) contains a very simple data table that represents the same data:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Fruit consumption</title>
</head>
<body>
<table summary="Fruit Consumption of under 15 year olds, March 2007">
  <caption>Fruit Consumption</caption>
  <thead>
    <tr><th scope="col">Fruit</th><th scope="col">Amount</th></tr>
  </thead>
  <tbody>
    <tr><td>Apples</td><td>10</td></tr>
    <tr><td>Oranges</td><td>58</td></tr>
    <tr><td>Pineapples</td><td>95</td></tr>
    <tr><td>Bananas</td><td>30</td></tr>
    <tr><td>Raisins</td><td>8</td></tr>
    <tr><td>Pears</td><td>63</td></tr>
  </tbody>
</table>
<p><a href="inlineimagelongdesc.html">Back to article</a></p>
</body>
</html>
```

The two different data representations side by side look like that seen in Figure 3.

Figure 3: You can link a document with complex data to an image by using the `longdesc` attribute.

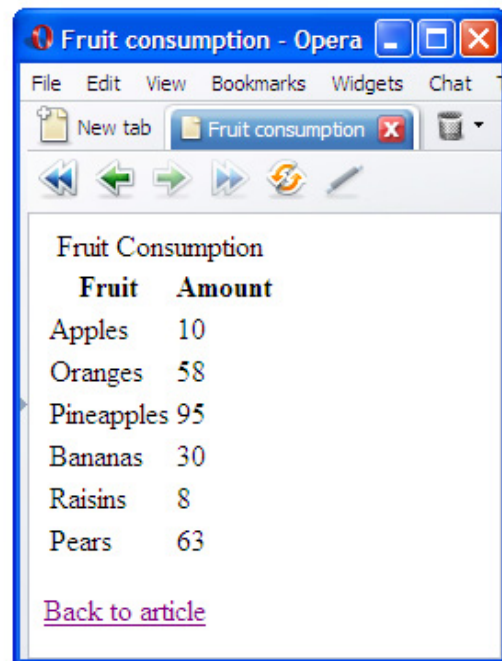
Note that there is no visual clue that there is a long description file connected with this image. Assistive technologies however will let their users know there is an alternative available.



Faster image display by defining the dimensions using width and height

When the user agent finds an `img` element in the HTML, it starts loading the image the `src` attribute points to. By default, it doesn't know the image's dimensions, so it'll just display all the text lumped together, then shift the rest of the document around when the images finally load and appear. This can slow down page loading and looks a bit confusing to the page visitors. To stop this happening you can tell the user agent to allocate the right amount of space for the images before they load by giving it the image's dimensions using the `width` and `height` attributes ([inlineimagewithdimensions.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.01//EN"
```



```
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Example of an inline image with dimensions</title>
</head>
<body>

</body>
</html>
```

This will display a place holder the size of the image until the image loads and takes up its place, therefore avoiding the unsightly page shift. You can also resize images using these attributes (try halving the attribute values in the above example, saving it, and then reloading the page), but this is not a good idea as the quality of resizing is not smooth in all browsers. It is especially bad to resize images to become thumbnails, as the idea of thumbnails is that you not only have a smaller image in physical size, but also in file size. Nobody wants to load a 300KB photo just to see a small image that could be 5KB.

So much for inline images

There are a lot more attributes you can use in images but most are deprecated as they define the layout and alignment of the image. This is not the job of HTML—it is what CSS was invented for. Suffice to say it is important to remember that images are—by default—inline elements. This means they can appear in between words in text without forcing new lines. This is great if you want to add small icons within copy, but it can be annoying when you are trying to create layouts using images and text. Using CSS you can

override the inline default and make images appear like block level elements (elements that appear on a new line when you add them to a document).

Background images with CSS

It is pretty safe to say that web design became a lot more fun when browsers started supporting CSS. Instead of hacking around in the HTML using table cells for positioning items on the page, non-breaking-spaces () to preserve spacing, and spacer GIFs (transparent 1x1 pixel GIF images that were resized to create margins) we can now use padding, margin, dimensions and positioning in CSS and leave the HTML free to just worry about the content structure.

CSS also means you can use background images in a very versatile way—you can position them behind and around your text any way you want, and also repeat images in regular patterns to create backgrounds. I'll only cover CSS images briefly here, as a later article will cover CSS background images in much more detail.

How to apply backgrounds with CSS

The CSS to apply images as backgrounds is pretty easy. Before you look at the CSS code below, load the [imagesandcss.html](#) example file in your browser, or look at Figure 4, to get an idea of all the different things that are possible with background images in CSS.

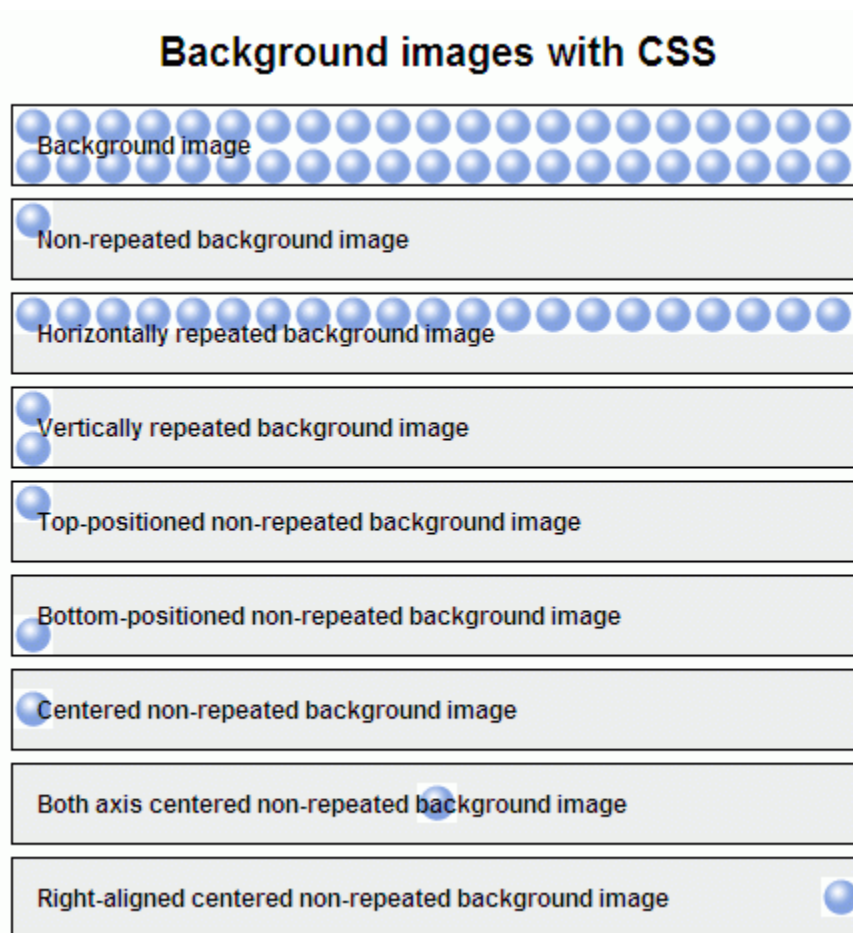


Figure 4: Backgrounds with CSS.

The different boxes are actually styled h2 heading elements with some padding and borders applied through CSS to give us enough space to show the background image. If you check out the HTML file, you'll see that each h2 element has a unique id so each one can have a different CSS rule applied to it. The CSS for the first example is the following:

```
background-  
image:url(ball.gif);  
background-color:#eee;
```

You add the image with the background-image selector and give it a URL in parenthesis to specify the image to be included. As a fallback in case the image is not available, you should also provide a background colour with the background-color selector and a (hexadecimal, named or RGB) colour value (in this case I

chose a light grey).

By default, background images will be repeated both horizontally and vertically to fill up the whole element space. You can however define a different repetition with the background-repeat selector:

- Don't repeat the image at all: `background-repeat:no-repeat;`
- Just repeat the image horizontally: `background-repeat:repeat-x;`
- Just repeat the image vertically: `background-repeat:repeat-y;`

By default the background image (if not repeated) will be positioned at the top and left corner of the element. You can however use `background-position` to move the background image around. The easiest values to choose are `top`, `center`, and `bottom` for the vertical alignment and `left`, `center`, and `right` for the horizontal alignment. For example, to position the image on the bottom right you need to use `background-position:bottom-right;`, while to centre the image vertically and apply it to the right you would use `background-position:center-right;`.

By controlling the repetition and the position of background images and using clever images you can create a lot of stunning effects that were not possible before CSS, and by keeping the background definitions in a separate CSS file you make it very easy to change the look and feel of a whole site by changing some lines of code. This will all be covered later in Article 30.

Summary

That's all you need to know to get you going when it comes to adding images to HTML. There are a lot more tricks available using images and CSS, but for now have a go with what you learnt here and concentrate on best-practice application of images. We talked about:

- The `img` element and its basic attributes:
 - `src` for the file location of the image
 - `alt` for text that should be available when the image is not loaded or cannot be seen
 - `title` for interesting (but not essential) additional information
 - `longdesc` to point to an external data file containing an alternative textual representation of the data illustrated in the image when the image is for example a complex chart
 - `width` and `height` to tell the browser how large the image is and therefore how much space to allocate for it
- The basics of CSS background images
 - When to use backgrounds (basically when the image does not need a text alternative but is only "eye candy" or "screen furniture", for layout.)
 - How to position and repeat background images in CSS

Exercise Questions

- Why is it important to add good text to an image in an `alt` attribute and do you really need one?
- If you have an image that is 1280x786 pixels large and you want to show a 40x30 pixel thumbnail, can you do that in HTML and is it wise to do so?
- What does the `longdesc` attribute do, and how do browsers show it?
- What do the `valign` and the `align` attributes do and why weren't they covered here?
- Where are CSS background images positioned inside an element by default, and how do they get repeated by default?

About the author



Photo credit: [Bluesmoon](#)

18: HTML links - let's build a web!

BY [CHRISTIAN HEILMANN](#) · 8 JUL, 2008

Introduction

In this tutorial you'll learn all about one of the most ground-breaking inventions in the history of the web—links. Links allow the reader of a document to follow them to another document and jump from server to server without having to disconnect and connect all over again. A lot has happened since they were first invented but one thing stayed the same: links are a very important part of the web experience and you can make it easy or hard for your web site's visitors depending on how you use them.

After you've gone through this article you'll know how to create links that are easy to understand and function in any environment. Furthermore you'll learn how linking affects your search engine popularity and you'll get some tips about wording links. As usual, [there is an accompanying zip file to this tutorial](#), which contains several files I'll refer to as we go along. The structure of the article is as follows:

- [What are links?](#)
- [The anatomy of an anchor link](#)
- [Link or target? The `name` and `href` attributes](#)
- [Don't leave any ambiguity about what you're linking to](#)
 - [Providing extra information with a `title` attribute](#)
 - [Linking to non-HTML resources—don't make people guess](#)
 - [External vs internal links](#)
- [Frames and popups—just say no](#)
- [Benefits of outbound and inbound links](#)
- [Link wording](#)
- [Link styling](#)
- [Summary](#)
- [Exercise questions](#)

What are links?

Links are parts of a web site (usually created using HTML, but not always) that point to other resources—other HTML documents, text files, PDFs, etc. There are links that should be followed automatically by the browser, created using `link` elements (you've already encountered some of those in earlier articles—they were used to import CSS files into an HTML document) and then there are links that are optional for the user to activate. These are called anchors and you can add them to the document using the `a` element.

The anatomy of an anchor link

You can turn any inline element in the document into an anchor link by adding an `a` element around it. For example, in the following HTML document the text *Yahoo Developer Network* gets turned into a link ([linkexample.html](#)).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Link Example</title>
<link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>A link to the YDN</h1>
  <p><a href="http://developer.yahoo.com">Yahoo Developer Network</a></p>
</body>
```

```
</html>
```

Visitors activating this link (either by clicking it with a mouse, or activating it with the keyboard or voice in some cases) will leave the current site and go to the YDN. There are more changes happening to the link itself, and we'll see about them later when we talk about link styling.

The anchor has several attributes you can add:

- `href`—the resource it points to (either an external file or an anchor ID).
- `id`—the anchor ID if the anchor is a target and not a link.
- `title`—extra information about the external resource.

Let's go through the most important attributes first and then talk about what you can do to make things easy to grasp for your visitors.

Link or target? The `id` and `href` attributes

An `a` element can play several roles depending on which attributes are set. The most common attribute you'll use is the `href` attribute, which defines what resource the link points to. This attribute can contain different values:

- A URL, which can be either in the same folder (`help.html`), relative to the current folder (for example `../../help/help.html`—2 dots means "go up one level in the site folder hierarchy"), absolute to the server root (for example `/help/help.html`—having a forward slash at the front of the address means the address starts at the root of the computer the page is on) or on a different server altogether (for example `http://wait-till-i.com` or `ftp://ftp.opera.com/` or `http://developer.yahoo.com/yui`).
- A fragment identifier or anchor name preceded by a hash (for example `#menu`). This points to a target inside the same document.
- A mixture of the first two—you can link directly to a section of a different document by pointing the `href` attribute to a URL followed by a fragment identifier (for example `http://developer.yahoo.com/yui/#cheatsheets`).

Any of these will make it a link as it points to somewhere else. On the other hand an `id` attribute will make it an anchor in the page—something another link points to. This is a bit confusing as both use the anchor element (`a`). To make it easy to remember think of it like this: an `id` attribute makes a link an anchor and you can use it to link to specific document sections. The following HTML has examples of all the different types of links in it ([linkexamples.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Different Link Example</title>
<link rel="stylesheet" href="linkexamplestyles.css">
</head>
<body>
  <h1>Different Link examples</h1>

  <h2>Example of in-page navigation with fragment identifiers, links and
anchors</h2>
  <div id="nav">
    <ul id="toc">
      <li><a href="#sec1">Section One</a></li>
      <li><a href="#sec2">Section Two</a></li>
      <li><a href="#sec3">Section Three</a></li>
      <li><a href="#sec4">Section Four</a></li>
      <li><a href="#sec5">Section Five</a></li>
    </ul>
```



```

</div>

<div id="content">
  <div>
    <h2><a id="sec1">Section #1</a></h2>
    <p><a href="#toc">Back to menu</a></p>
  </div>
  <div>
    <h2><a id="sec2">Section #2</a></h2>
    <p><a href="#toc">Back to menu</a></p>
  </div>
  <div>
    <h2><a id="sec3">Section #3</a></h2>
    <p><a href="#toc">Back to menu</a></p>
  </div>
  <div>
    <h2><a id="sec4">Section #4</a></h2>
    <p><a href="#toc">Back to menu</a></p>
  </div>
  <div>
    <h2><a id="sec5">Section #5</a></h2>
    <p><a href="#toc">Back to menu</a></p>
  </div>
</div>

<h2>Some other link examples</h2>
<ul>
  <li><a href="http://developer.yahoo.com">Yahoo Developer Network</a></li>
  <li><a href="http://dev.opera.com/articles/view/the-freelancing-business-part-1-pricing/#marketing">Tips on marketing yourself</a></li>
  <li><a href="ftp://get.opera.com/pub/opera/win/">Download different Opera versions</a></li>
  <li><a href="http://farm1.static.flickr.com/56/188791635_0b8bdd808d.jpg?v=0">Photo of my book</a></li>
</ul>

</body>
</html>

```

Open this file in your browser of choice and experiment with it. You'll find that activating any of the links in the first list will jump to the appropriate section of the document. This is because they are connected by the same fragment identifier—the first link in the list for example has an `href` attribute of `#sec1`, which is the same as the ID value of the link inside the first `h2` element of the content. This is all you need to do to connect two anchor elements in a document—use the same value preceded by a hash if you link to it in an `href` attribute. You might also have realized that the URL in the location bar of your browser changed and now shows the fragment identifier at the end of it, which means visitors can bookmark this section or email the link to other people to send them exactly where they should go.

However, if you activate any of the “Back to menu” links, the same functionality occurs. How is that possible? Fragment identifiers can be any element with an ID. To recap:

- anchor links can have a fragment identifier as the value of the `ref` attribute—this fragment identifier must start with a hash sign (#).
- When activated, this link will jump to any HTML element with an `id` of this value. The IDs on each page must be unique.
- IDs follow certain naming conventions. Most importantly is that they must start with an alphanumeric character and must not have any spaces in them.

That covers the menu and the different sections in the example but what about the other links? If you try them out you'll see that they point to different targets—one goes to another site, another displays a photo

and the third one shows a specific section of another web page (found by jumping to a specific ID). If all of that worked for you, great—but what if you or your browser couldn't understand some of these resources?

Don't leave any ambiguity about what you're linking to

The most important thing to remember about links is that they are a substantial part of your relationship with your visitors. They trust that when you offer them a link, they can follow it and get good, relevant information. If your links don't work because the linked resource is not available or in a format the visitor cannot consume you betrayed that trust and lost credibility. Don't let that happen.

Providing extra information with a `title` attribute

Like almost any other HTML element you can add a `title` attribute to an `a` element to add some extra information. Browsers will show a so called tooltip when visitors hover their mouse cursor over the link. This tooltip then tells them what the link is about. For example you could give a small introduction to the content and location of the linked document ([titleexample.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Adding extra information with a title attribute</title>
<link rel="stylesheet" href="linkexamplestyles.css">
</head>
<body>
  <h1>Adding extra information with a title attribute</h1>
  <ul>
    <li>Find more information on the <a title="The Yahoo Developer Network is the
main hub for all the developer tools Yahoo offers, including the Yahoo User
Interface Library (YUI) and the Design Patterns repository"
href="http://developer.yahoo.com">Yahoo Developer Network</a>.</li>
  </ul>

  </body>
</html>
```

Figure 1: Adding a `title` attribute shows the information as a tooltip when visitors hover over the link.

However, you cannot expect visitors to have enough patience and hand-eye coordination to rely on this for crucial information. Visually impaired users, who cannot see the page at all, are very likely not to be able to reach this information. While screen readers have the option to read out `title` attributes for the end user it is turned off by default—which is why you should never use the `title` attribute for crucial information about the link. Crucial information might be:

- Linking to non-HTML resources like PDF files, images, videos, sound files or downloads.
- Leaving the current site and linking to another server (external vs internal links).
- Linking to a document that'll open in a different frame or a popup.

Linking to non-HTML resources—don't make people guess

It can be very annoying when you click on a link and your browser does not know what to do with what the link you clicked on points to. It is pretty common however to see web sites link to images, PDF documents and videos without warning their visitors to be prepared. Videos especially are very often a cause for browser crashes. Furthermore, the resource might be on the larger side (20MB PDF anyone?) which means that visitors might prefer to download it rather than opening it inside the browser and add to its already hefty memory consumption, or just not access it at all.

One of the biggest success factors of a web product is not keeping people guessing what happens when they perform an action and instead tell them flat out what effects their action will have. In the case of

links all you need to do to prevent a lot of frustration is to tell your visitors what the linked resource is. Here are some examples (linkingnonhtml.html):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Linking non-HTML resources</title>
<link rel="stylesheet" href="linkexamplestyles.css">
</head>
<body>
  <h1>Linking non-HTML resources</h1>

  <ul>
    <li>Find more information on the <a href="http://developer.yahoo.com">Yahoo
Developer Network site (external)</a></li>
    <li>Download the <a href="http://www.wait-till-i.com/stuff/JavaScript-DOM-
Cheatsheet.pdf">Dom Cheatsheet (PDF, 85KB)</a></li>
    <li>Pick and <a href="ftp://get.opera.com/pub/opera/win/">download different
Opera versions from their FTP (external)</a></li>
    <li>Check out a <a
href="http://farm1.static.flickr.com/56/188791635_0b8bdd808d.jpg?v=0">Photo of my
book (JPG, 200KB)</a></li>
  </ul>

</body>
</html>
```

By providing such information about the linked file and its size you leave the decision of what to do with it with your visitors rather than expecting them to have certain browser settings or installed software. If you mix that with clever styling you can even make it look pretty and intuitive. If you want to be very safe, also offer a help section that explains what the different file formats are and where you could get the software needed to display them.

External vs. internal links

One of the biggest fears of business decision makers when it comes to their company's web sites is people leaving prematurely. This is often the reason for never offering third party links (unless the third parties pay money for the privilege of having web traffic directed towards them). I'll come back to this error in judgment later on; for now let's talk about what people do to avoid visitors leaving their site and how these measures affect the site's success.

Frames and popups—just say no

The fear of losing visitors to other sites while still wanting to link to them gave us some inventions in web development that have been a thorn in the side of usable sites for years: frames and popups.

Using HTML frames means you separate the page shown in the browser into several different documents. The benefit is that the document seemingly stays the same even when you load parts of it either from your own server or from third party servers. This is where the usefulness ends however—frames are a terrible user experience and actually harmful:

- Search engines can never index a whole page but instead might show up parts of a page in search results that don't make sense out of context.
- Visitors cannot bookmark the page—the next time they open their bookmark they'll get the initial state of the frameset and not the page as they left it.
- Visitors dependent on assistive technology have a very hard time navigating around framesets.
- Third party sites might not like to be shown inside a frameset and use "framebreaker" scripts that replace framesets with the real URL when you try to embed them. This is to stop criminals luring

Internet users into entering for example credit card information into a web site that seemingly looks like a bank (so called “phishing”).

Links inside a frameset use the `target` attribute of the anchor to target the correct frame. Each frame in a frameset gets a certain name and activating the link would open the document defined in the `href` attribute in that frame. If the frameset is not available (for example when a visitor found the document with the links via a search engine) each link opens in a new browser instance.

Opening a new browser instance is another common way to link to third party sites—either with a scripted pop-up window or with a `target` attribute with the value of `_blank`. The fact that every modern browser comes with a pop-up blocker should give an indication of how safe it is to rely on this technique in this day and age. It’s not!

In short: do not use the `target` attribute when you create links, unless you really know what you are doing. It is an outdated idea anyhow—these days most browsers have tabbed interfaces, so users can open third party sites in the background for later reading while they stay on your site. Under certain circumstances you may want to indicate the difference between external and internal links, but always leave it to the discretion of the visitor what they want to do with them.

Benefits of outbound and inbound links

There are several good reasons for linking to third party sites even when they are competitors.

- It gives you credibility in the eyes of your visitors—you are so sure of the quality of your content that you don’t shy away from the competition.
- It is an opportunity to deliver a full service—you can link to content and articles or even products on other sites that you don’t cover but that might be of interest for those visitors who want to dive deeper into the topic at hand.
- You can prove a point by building on an older article by a third party and offering a better or different solution and referencing the old article as a proof via a link.

The usefulness of inbound links (links pointing from a third party site to yours) is less debated. The more often that valid and high quality sites link to yours with relevant keywords, the higher you’ll rank in search engines such as Google. Before that happens however you need to prove that you don’t shy away from linking to others either.

The relevant keywords bring us to another very important part of creating good links: how to word them.

Link wording

I’ve covered this partly in the section about linking to non-HTML resources, but it is good to remind ourselves that links are not only part of the page copy but also interactive elements in the document.

Some assistive technologies will offer a list of links instead of the whole document to allow visitors to quickly navigate their way through it, which means that you need to make sure that your link texts make sense outside the context they are in. You can easily check this in Opera by opening any web site and choosing Tools > Links from the menu or pressing `Ctrl + Shift + L`. You’ll get a new tab that shows all the links in the document and where they point to.

This means you should make sure not only that all the link texts make sense, but also that there are not links that have the same wording but point to different resources. The classic mistake here is “click here” links, worded for example like “To download the latest version of our tool click here”. It is much better to use a link text that explains what it points to—in this case “You can download the latest version of our tool and try it out for yourself”.

The same applies to “more” links. You’ll find these in news sites where you get a heading and a teaser text and a “more” or “full story” link to follow. The solution to this problem is to either use a linked “more” image and give it a unique alternative text or to add a span inside the link after the “more” and hide it

with CSS. You'll hear more about these tricks in the tutorial about menus and navigation later in the course.

Link styling

We're not quite at CSS yet with this course, but it is useful to consider at this point that the way links look is very important and there are several different link states to consider. The link states (which later on relate to CSS pseudo-selectors—this sounds complex, but it isn't) are:

- `link`—the default state—it defines what links should look like in a certain part of the document. By default, unvisited links are coloured blue.
- `visited`—the style of a link that was already visited before (and might already be in the browser cache). By default, already visited links are colored purple.
- `hover`—the style of a link whilst the mouse cursor is hovering over it.
- `active`—the style of the link while it is activated, ie while the connection to the other site is in progress; it is also the style of the last activated link when you arrive at the document by going back in your browser.

Summary

We covered a lot this time, but it is very important to remember how links work and what they should do. You will learn a lot of tricks and techniques in your career as a web developer to override this default behaviour and I hope you'll stop and wonder if what you are trying to do is really necessary.

I talked about:

- The anatomy of the `a` element and its (non-deprecated) attributes
- The difference between `a` elements as links (with an `href` attribute) and as anchors (with a `name` attribute)
- The need for an anchor name to be unique
- The necessity to tell visitors what to expect when they follow a link (what format is the file and how big is it)
- How to add information via the title attribute that gets displayed as a tool-tip - and why this is not a safe way of providing crucial information
- The difference between external (pointing to third party sites) and internal (pointing to documents on the same server) links
- Outdated practices like popups and frames and why you should avoid them
- The benefits of linking to, and getting linked to by, other sites
- How to word links properly so that they make sense out of context, and why this is necessary
- The background behind basic link styling.

With this knowledge under your belt you should be able to write HTML documents that link properly and you are ready to go and think about menus and site navigation.

Exercise questions

- What is wrong with the following link: `get our latest report?`
- What is the `target` attribute in links for and are there any good uses for it?
- I've talked about link relationships and connections between links and anchors. Is there an attribute for links that describes relationships between documents, too?
- How can you write a link that sends the visitors to an element further down the page when they click it? What do you need to make sure of beforehand?

About the author



Photo credit: [Bluesmoon](#)

19: HTML tables

BY [JENIFER HANEN](#) · 8 JUL, 2008

Introduction

“Ack!”—how do you use web standards to organize reams of data? The very idea of tons of nested elements needed to get all the data into nice little rows and boxes ought to put your brain into “Ack!” mode, but there is a solution—tables to the rescue!

In web design tables are a good way to organize data into a tabular form. In other words, think of tables, charts, and other information graphics that help you to see and process a great deal of information in a summary that allows you to compare and contrast different pieces of data. You see them all the time on websites, whether they are giving a summary or comparison of political election results, sports statistics, price comparisons, size charts, or other data.

Back in the Jurassic Age of the Internet before CSS was popularised as a method of separating the presentation from structure of the HTML, tables were used as a way to lay out web pages—to create columns, boxes, and generally arrange the content. This is the wrong way to go about it; tables for layout resulted in bloated, messy pages with tons of nested tables—larger file sizes, hard to maintain, hard to modify after the fact. You’ll still see sites like this on the Web, but rest assured that these days you should just use tables for what they are designed for—tabular data—and use CSS to control layout.

In this article I will cover how to use HTML tables properly—the structure is as follows:

- [The most basic table](#)
- [Adding some more features](#)
- [Structuring the table further](#)
- [CSS to the rescue: a better looking table](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

The most basic table

I will start with the semantic HTML code required to render a basic table—this particular example compares recent volcanic eruptions in the Pacific region of North America. I like volcanos and when I was a kid, I convinced my mom to take me to several of these volcanos on our summer road trips to visit Grandma. I dearly hoped that one of the [Pacific Northwest volcanos](#) would erupt while we were on holiday, but to no avail. The first table looks like so:

```
<table>
  <tr>
    <td>Volcano Name</td>
    <td>Location</td>
    <td>Last Major Eruption</td>
    <td>Type of Eruption</td>
  </tr>
  <tr>
    <td>Mt. Lassen</td>
    <td>California</td>
    <td>1914-17</td>
    <td>Explosive Eruption</td>
  </tr>
  <tr>
    <td>Mt. Hood</td>
    <td>Oregon</td>
    <td>1790s</td>
    <td>Pyroclastic flows and Mudflows</td>
  </tr>
</table>
```

```
</tr>
<tr>
  <td>Mt .St. Helens</td>
  <td>Washington</td>
  <td>1980</td>
  <td>Explosive Eruption</td>
</tr>
</table>
```

This code renders as follows:

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt .St. Helens	Washington	1980	Explosive Eruption

Let's start by breaking down the HTML markup used in the above code:

- `<table></table>`: The table is necessary to indicate to the browser that you wish to arrange the content in a tabular fashion.
- `<tr></tr>`: The `tr` element establishes a table row. This allows the browser to organize any content between the `<tr>` and `</tr>` tags in a horizontal fashion, or all in a row.
- `<td></td>`: The `td` element defines the table cell or each individual space for content within the row. Only use as many `td` table cells as needed for actual data. Don't use empty `td` cells for white space or padding—you use CSS to create any white space or padding needed, as this is not only a good way to separate presentation from structure but it also makes the table easier to understand for people with visual impairments who are using screenreaders to read the table contents out to them.

Note that the basic elements must be nested as follows:

```
<table>
  <tr>
    <td>content</td>
    <td>content</td>
    <td>content</td>
  </tr>
</table>
```

To order them in another fashion will cause the browser to spit up the equivalent of an internet hair ball and render the table in a very odd fashion if at all.

Adding some more features

Now the basic table is in place, you can add some slightly more complex table features—first, I will add a caption and Table headers to help improve the data both in terms of semantics and legibility for screen readers. The updated table markup looks like so:

```
<table>
  <caption>Recent Major Volcanic Eruptions in the Pacific Northwest</caption>
  <tr>
    <th>Volcano Name</th>
    <th>Location</th>
    <th>Last Major Eruption</th>
    <th>Type of Eruption</th>
  </tr>
```

```
<tr>
  <td>Mt. Lassen</td>
  <td>California</td>
  <td>1914-17</td>
  <td>Explosive Eruption</td>
</tr>
<tr>
  <td>Mt. Hood</td>
  <td>Oregon</td>
  <td>1790s</td>
  <td>Pyroclastic flows and Mudflows</td>
</tr>
<tr>
  <td>Mt. St. Helens</td>
  <td>Washington</td>
  <td>1980</td>
  <td>Explosive Eruption</td>
</tr>
</table>
```

This code is rendered as:

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption

The new elements used here are:

- `<caption></caption>`: The `caption` element allows you to title the table data. Most browsers will center the caption and render it the same width as the table, unless one chooses to use CSS to align the text differently.
- `<th></th>`: The `th` element delineates the content between the tag as the table head titles for each column. This is useful not just to help semantically describe what the function of this content is, but it also helps render it more accurately in a variety of browsers and devices. The above example is the most stripped down way to use the `th` element.

Structuring the table further

As a final step in structuring our table, I will define header and body table sections, add a footer and define the scope of row and column headings. I will also add a summary attribute to describe the table contents. The final markup looks like so:

```
<table summary="a summary of recent major volcanic eruptions in the Pacific Northwest">
  <caption>Recent Major Volcanic Eruptions in the Pacific Northwest</caption>
  <thead>
    <tr>
      <th scope="col">Volcano Name</th>
      <th scope="col">Location</th>
      <th scope="col">Last Major Eruption</th>
      <th scope="col">Type of Eruption</th>
    </tr>
```

```

</thead>
<tfoot>
  <tr>
    <td colspan="4">Compiled in 2008 by Ms Jen</td>
  </tr>
</tfoot>
<tbody>
  <tr>
    <th scope="row">Mt. Lassen</th>
    <td>California</td>
    <td>1914-17</td>
    <td>Explosive Eruption</td>
  </tr>
  <tr>
    <th scope="row">Mt. Hood</th>
    <td>Oregon</td>
    <td>1790s</td>
    <td>Pyroclastic flows and Mudflows</td>
  </tr>
  <tr>
    <th scope="row">Mt. St. Helens</th>
    <td>Washington</td>
    <td>1980</td>
    <td>Explosive Eruption</td>
  </tr>
</tbody>
</table>

```

This table code looks like so in a browser:

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Compiled in 2008 by Ms Jen			
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption

The new elements/attributes are as follows:

- The `thead`, `tbody` and `tfoot` elements: These define the table's header, body and footer respectively. Unless you are coding a really complex table with many columns and rows of data, using these may be overkill. In a complex table, however, using them can not only structure the content for the coder's sake, but also for browsers and other devices.
- The `colspan` and `rowspan` attributes: The `colspan` attribute creates a table cell that will span more than one column. In the case of the above footer, I wanted the one table cell to span the whole width of the table, thus I told it that it was to span four columns. Alternately, you can add a table cell `rowspan` attribute that will allow the table cell to span x amount of rows, for example `<td rowspan="3">`.
- The `summary` attribute: This attribute is used to define a summary of the table contents, for use by screenreaders (notice that you didn't see it on the rendered version of the table above.) Note that in the older WC3 recommendations, WCAG 1.0 and HTML 4.0, it says you can use the `summary` attribute as detailed above. In newer drafts of the specs however, the `summary` attribute is not mentioned. Whether to still use the `summary` attribute seems undecided, so for now we at the Web

Standards Curriculum have agreed that it is safe to still use it. After all, it doesn't cause anything to break, and it confers accessibility advantages.

- The `scope` attribute: You may also have noticed the `scope` attributes in the `th` tags, and the fact that I have defined the volcano names as headings too, inside the data rows! This is perfectly allowable, but anyway, I digress. The `scope` attribute can be used in the `th` element to tell screen readers that the `th` content is the title for a column or a row. The `scope` attribute is only used in the `th` element.

CSS to the rescue: a better looking table

The above listed elements and attributes are all that is necessary to code a good data table. Now the HTML structure is in place, let's look at some simple CSS to make the table look a bit nicer:

```
body {
    background: #ffffff;
    margin: 0;
    padding: 20px;
    line-height: 1.4em;
    font-family: tahoma, arial, sans-serif;
    font-size: 62.5%;
}

table {
    width: 80%;
    margin: 0;
    background: #FFFFFF;
    border: 1px solid #333333;
    border-collapse: collapse;
}

td, th {
    border-bottom: 1px solid #333333;
    padding: 6px 16px;
    text-align: left;
}

th {
    background: #EEEEEE;
}

caption {
    background: #E0E0E0;
    margin: 0;
    border: 1px solid #333333;
    border-bottom: none;
    padding: 6px 16px;
    font-weight: bold;
}
```

When applied to our final table markup, the table looks as seen in Figure 1. You can also [check out the example live here](#).

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 1: The table now looks a lot more visually appealing.

Oooh... Look, much better. You can choose to style the table anyway you want, but the above provides a baseline to work with. You'll learn a lot more about styling tables with CSS in a later article (to be published soon), but for now I'll just briefly break down what each section of this CSS is doing:

- `body`: In the above CSS, I have added properties to set the margin (to zero in this case), padding to give a little room, background color (white), font size and family, as well as the line height to also give a little breathing room. You can [download the code for this example here](#)—try altering the properties in the CSS file to see how things change.
- `table`: borders have been added using the CSS border declaration. To make this work correctly, I also had to reset the `table` selector above with the `border-collapse` property set to `collapse`, which will not only reset the border values in the table but also allow the `border-bottom` to be a straight rule line across the whole row rather than being broken up at the end of each table cell. I chose a width of 80% for this example.
- `th` and `td`: In the above example CSS, I have set the text alignment to be left, but you could set it to center or even give the various `th` and `td` elements class names and then use the CSS to have more control over each row or column (in the row case, you would give the `tr` element tag a class name). I also gave both the `th` and `td` a bit of padding to open up the rows and allow for greater readability. In the case of the `th` selector above, I set another color as to differentiate the headings from the rest of the table.
- `caption`: If you do not set CSS properties for the `caption` selector, then it does not have a border and is the same background color as the whole page even though the HTML markup for the caption is within the `table` tag. Thus, in the above example I have given the caption a border (with no bottom border, as the border in the table already provided it), a different background color and bold type to set the caption apart from the table header row.

Summary

In this article I have gone through all you need to know to create effective HTML data tables. That's a wrap! I'll leave you with some pertinent thoughts:

- It is important that tables are correctly coded to be readable by a variety web browsers, mobile, accessible, and other devices. Table HTML is best kept to a minimum, and you should use CSS to style the tables. You'll learn a lot more about CSS later on in the course.
- Tables can be accessible to mobile devices and users that use screen reading software by keeping the code clean, using attributes such as `scope` and `summary` as well as the `caption` element to help announce clearly and semantically what the respective sections are for. Also important for accessibility is to not use empty table cells for spacing (use CSS for this instead).

Further reading

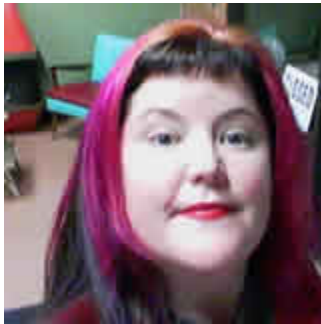
- [W3C HTML 4 Tables Recommendation](#)

- [W3C CSS 2 tables recommendation](#)
- [Roger Johansson's "Bring on the Tables"](#)

Exercise questions

- Start by coding a simple table with only the 3 main table elements: `table`, `tr`, and `td`. Save it and view it in a browser.
- Much like the second example above, add a caption, header, and footer to your table. How does that change what you see in the browser?
- What can you do to make your table more accessible to screen readers and hand held devices?
- Now create a CSS file. Try styling the borders, padding, and cell spacing of your table with only CSS and no attributes in your HTML markup. Add background color and style the fonts.
Have fun.

About the author



A web designer/developer by trade, a photographer, moblogger and professional art weirdo for the love of it, Ms. Jen started her art and design career in the high chair with her love of personal food art and food wall art at the age of 11 months. She taught herself HTML in 1996 when a computer snob said that an artist could not learn to code and has been fully in love with all things web design ever since.

Ms. Jen is the owner and founder of Black Phoebe Designs, a web & mobile design firm. Ms. Jen has has a Masters degree in Computer Science and Multimedia Systems from Trinity College in Dublin, Ireland, and taught web design at an LA area university from 2001-2005. She has participated in two Nokia mobile blogging projects, Wasabi Lifeblog (2004-2005) and the Nokia Urbanista Diaries (2008). Ms. Jen can always be found online at blackphoebe.com or blackphoebe.mobi.

20: HTML forms—the basics

BY [JENIFER HANEN](#) · 8 JUL, 2008

Introduction

Everyone has seen a form. Everyone has used one. But have you coded one?

A form online is any area where you can input information into a web page, for example entering text or numbers into a text box, checking a tick box, causing a radio button to “fill in”, or selecting an option from a list. The form is then submitted to the web site when you push the submit button.

You’ll see forms used on the Web everywhere, for entering user names and passwords on login screens, commenting on blogs, filling your profile in on a social networking site, or specifying billing information on a shopping site.

It is easy to create a form, but what about a web standards compliant form? By now, if you have been working through the Opera Web Standards Curriculum, you are hopefully convinced that web standards are the way to proceed forward. The code required to create a standards compliant accessible form are no more work to implement than a sloppy form.

So, let’s start with the most basic and simple form that one could possibly want to use and build our way up in complexity after that—in this article I’ll cover all the basics you need to know to create elegant, accessible form structures with HTML. The article structure is as follows:

- [Step one: The basic code](#)
- [Step two: Adding structure and behavior](#)
- [Step three: Adding semantics, style and a bit more structure](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

Step one: The basic code

Let’s start off with a really basic comment form, the sort of form you would use on a web site to allow people to give you feedback on something such as an article you’ve written, or allow someone to contact you without knowing your e-mail address. The code looks like this:

```
<form>
Name: <input type="text" name="name" id="name" value="" />
Email: <input type="text" name="email" id="email" value="" />
Comments: <textarea name="comments" id="comments" cols="25" rows="3"></textarea>
<input type="submit" value="submit" />
</form>
```

If you enter this into an HTML document and then open that document in a browser, the code is rendered as shown in Figure 1.



Figure 1: The first, basic form example.

Try it for yourself—enter the above code into your own sample HTML document and load it in a browser, or [click here to navigate to the form in a separate page](#). Try playing around with the different form controls to see what you can do with them.

As you read the code, you'll see an opening `<form>` tag, a `</form>` closing tag, and some bits in between the two. The element contains two text inputs in which the page's reader can enter their name and email address, and a textarea that can be filled with a comment to submit to the site owner.

What have we got here?

- `<form></form>`: These two tags are essential to start and end a form—without them you don't have a web form. Every form must start and end with `<form>` and `</form>` tags.

The `<form>` tag can have a few attributes, which will be explained in Step Two, but please do note that you can't nest forms inside each other.

- `<input>` (should be `<input />` if you're using an XHTML doc type): This tag defines the area where the reader / user can insert the information. In our case above, the input tag defines the text boxes where the reader can type in their name and email.

Every input tag must have a `type` attribute to define what it will receive: the possible attribute values are `text`, `button`, `checkbox`, `file`, `hidden`, `image`, `password`, `radio`, `reset` or `submit`.

Every `<input>` tag must also have a `name`, which you the coder can decide on. The `name` attribute informs the thing that the data is sent to when the form is submitted (be it a database, or an email sent to the site's administrator via a server-side script) what the name of the information in the input box is. When the form is submitted, most scripts use the `name` attribute to place the form data into a database or into an email that can be read by a person.

Thus, if the `<input>` element is for the reader / user to enter their name into, then the `name` attribute would be `name="name"` or `name = "first name"`, etc. If the `<input>` tag is for an email address, then the `name` attribute would be `name="email"`. To make it easier on yourself, and the person who will use the form, it is recommended that you name the `<input>` element in a semantic fashion.

By semantically, I mean naming it according to what it's function is, as detailed above. If the input is to receive an email address, then name it `name="email"`. If it is to be the street address of the reader / user, then name it `name="street-address"`. The more accurate the word usage the easier it is not only for you to code and then perform maintenance tasks on at a later date, but also for the person or database receiving the form. Think lean and mean with accurate meaning.

- Every `<input>` tag should also have a `value` attribute. The value can be set to blank—`value=""`—which will tell the processing script to just insert whatever the reader / user types into the box. In the case of a checkbox, radio button, hidden, submit, or other type attributes you can set the value to equal what you want the final input to read. Examples: `value="yes"` for yes, `value="submit"` for a submit button, `value="reset"` for a reset button, `value="http://www.opera.com"` for a hidden redirect, etc.

Examples of how to use the `value` attribute:

A blank value attribute, which the user input determines the value of:

- The code says: `<input type="text" name="first-name" id="first-name" value="" />`
- The user inputs: Jenifer
- The value of `first-name` is sent as "Jenifer" when the form is submitted.

A predetermined value:

- The code says: `<input type="checkbox" name="mailing-list" id="mailing-list" value="yes" />`

- The user checks the box as they wish to join the website’s mailing list.
- The value of `mailing-list` is sent as “yes” when the form is submitted.
- After the two `<input>` elements, you can see something a bit different—the `textarea` element.

The folks at `textarea` bring you a nice, new, improved space to input text into. Not your ordinary, plain old one line text box that our friend `<input>` provides, the `textarea` element provides multiple lines of input, and you can even define how many lines are available to enter text into. Note the `cols` and `rows` attributes—these are required for every `textarea` element, and specify how many columns and rows big to make the text area. The values are measured in characters.
- Last but not least, you have a special `<input>` element with the attribute `value="submit"`. Instead of rendering a one line text box for input, the submit input will render a submit button that, when clicked, submits the form to whichever target the form has specified to send it’s data to (currently this isn’t defined at all, so submitting the form will do nothing.)

Step two: Adding structure and behavior

So, you clicked on the form #1 link above, filled it out and clicked Submit—why didn’t it do anything, and why does it look so bad and all in one line? The answer is that you haven’t structured it yet, or defined a place for the data the form is collecting to be submitted to.

Let’s go back to the drawing board, with a new form:

```
<form id="contact-form" action="script.php" method="post">
  <input type="hidden" name="redirect" value="http://www.opera.com" />
  <ul>
    <li>
      <label for="name">Name:</label>
      <input type="text" name="name" id="name" value="" />
    </li>
    <li>
      <label for="email">Email:</label>
      <input type="text" name="email" id="email" value="" />
    </li>
    <li>
      <label for="comments">Comments:</label>
      <textarea name="comments" id="comments" cols="25"
rows="3"></textarea>
    </li>
    <li>
      <input type="submit" value="submit" />
      <input type="reset" value="reset" />
    </li>
  </ul>
</form>
```

This form looks like Figure 2 when rendered in a browser:



Figure 2: The second form example—looking better, but still not perfect.

You can [play with the improved form on a separate page by clicking here](#).

Here I have made a few additions to the basic, simple form. Let's break it down so you know what I did:

- There are some new attributes inside the `<form>` tag. I added an `id` attribute to not only semantically name what this form is called, but also to provide a unique ID to identify the form so it can be more easily styled using CSS or manipulated using JavaScript if required. You can only have one of each `id` per page; in this case I called it `contact-form`.
- Lights, camera, action! When you pressed the submit button in the first form and it did not do anything, this was because it had no action or method. The `method` attribute specifies how the data is sent to the script that will process it. The two most common methods are "GET" & "POST". The "GET" method will send the data in the browser's URL. Unless you have a specific reason to use "GET", it is probably best to not use it if you are trying to send secure information as anyone can see the information as it is transmitted via the URL. "POST" sends the data via the script that powers the form, either to an email that is sent to the site's administrator, or a database to be stored and accessed later, rather than in the "GET" URL. ["POST" is the more secure and usually the better option](#).

If you are very concerned about the security of the data in the form, for example if you are submitting a credit card number to a shopping site, then you should use the https protocol with a secure socket layer (SSL). Basically, this means that data will be sent over the https protocol, not the http protocol. Have a look at the URLs next time you are paying for something on a shopping site, or using online banking—you'll probably see https:// in your address bar, not http://. The difference is that an https connection is a bit slower to transmit than http, but the data is encrypted, so anyone hacking into the data connection can't make any sense out of it while it is in transit. Talk to your web host for information on how they can provide you with https and SSL.

- The `action` attribute specifies what script file the form data should be sent to for processing. Many web hosts will have a generic send mail script or other form scripts available for use (see your host's documentation for information) that they have customized to their servers. On the other hand, you could use a server-side script that you or someone else has created to power your form. Most of the time, folks use languages such as PHP, Perl or Ruby to create a script that will process the form—you could for example send an email containing the form information, or input the form information into a database to be stored for later use.

It is outside of the scope of this course to write up a server-side script for you, or teach you how to write server-side code yourself—please inquire with your host to find out what they offer, or find a nice programmer to befriend.

Here are a few resources to get you started if you would like to investigate server-side scripting:

- Perl: <http://www.perl.com/>
- PHP: <http://www.php.net>
- PHP documentation on Forms: <http://uk3.php.net/manual/en/tutorial.forms.php>
- Python: <http://python.org/>
- Ruby: <http://www.ruby-lang.org>
- Sendmail: <http://www.sendmail.org/>
- ASP.NET: <http://www.asp.net/>
- The second line that's been added to our Step Two form is the "hidden" input field—this is a redirect. What?

Under the goal of separating markup structure from presentation and behavior, it is ideal to use the script that will power the form to also redirect the user once the form is submitted. You don't want your users to be left sitting there looking at the form page, wondering what the heck to do next after they've submitted the form; I'm sure you'll agree that it is a much better user experience to instead redirect your users to a thank you page featuring "what to do next" links, after a successful form submission. This line in particular specifies that after this form is submitted, the user will be redirected to the Opera homepage.

- To improve the look of the form, I have put all the form elements into an unordered list so that I can use the markup to line them up cleanly and then use CSS to polish the look.

Some folk would argue that you should not use an unordered list to markup a form, but use a definition list instead. Others would argue that one should not use a list at all but use CSS to style the `<label>` and `<input>` tags. I will let you research this debate and make up your own mind on which is more semantically correct. For our simple exercise I will use the unordered list.

- Last but not least in step two, I've labeled the form elements. Both in terms of meaning and making the form accessible to a wide range of internet enabled devices, it is best to give all the form elements labels—check out the contents of the `label` elements - these labels are tied to their respective form elements by giving the `input` and `textarea` elements `ids` that have the same value as the labels' `for` attributes. This is great because it not only gives a visual indicator of the purpose of each form field on the screen, but it also gives the form fields more meaning semantically. For example, a visually impaired user using the page with a screenreader can now see which label goes with which form element. The `ids` can also be used to style individual form fields using CSS.

The 2nd form displays a bit better, but it has been beaten with the default ugly stick. Time to add a few more bits and bobs before applying some style.

Step three: Adding semantics, style and a bit more structure

Now I'll finish off what I started at the beginning of the article, with the following final version of my example form:

```
<form id="contact-form" action="script.php" method="post">
  <fieldset>
    <legend>Contact Us:</legend>
    <ul>
      <li>
        <label for="name">Name:</label>
        <input type="text" name="name" id="name" value="" />
      </li>
      <li>
        <label for="email">Email:</label>
        <input type="text" name="email" id="email" value="" />
      </li>
      <li>
        <label for="email">Comments:</label>
        <textarea name="comments" id="comments" cols="25" rows="3"></textarea>
      </li>
      <li>
        <label for="mailing-list">Would you like to sign up for our mailing
list?</label>
        <input type="checkbox" checked="checked" id="mailing-list" value="Yes,
sign me up!" />
      </li>
      <li>
        <input type="submit" value="submit" />
        <input type="reset" value="reset" />
      </li>
    </ul>
  </fieldset>
```



```
</form>
```

When rendered in a browser, this form looks as shown in Figure 3.

Figure 3: The final form example in all its glory.

To see this form live in a browser and play with it, [following this link](#) to the form on a separate page.

The last two major elements I have added to this form are `fieldset` and `legend`. Both of these elements are not mandatory, but are very useful for more complex forms and for presentation.



The `fieldset` element allows you to organize the form into semantic modules. In a more complex form, you could for example use different `fieldsets` to contain address information, billing information, customer preference information, and so on. The `legend` element allows you to name each `fieldset` section.

I've also applied a little bit of CSS to this form, to style the structural markup. This is applied to the third form example using an external stylesheet—[click on this link to see the styles](#). The two most important tasks I wanted the basic CSS to do is add margins to line up the labels and input boxes, and get rid of the unordered list's bullet points. Here is the CSS that resides in the external stylesheet:

```
#contact-form fieldset {width:40%;}
#contact-form li {margin:10px; list-style: none;}
#contact-form input {margin-left:45px; text-align: left;}
#contact-form textarea {margin-left:10px; text-align: left;}
```

What does it do? The first line styles the `fieldset` border to not take up the whole page; you could also set the border to none using `{border: none;}` if you didn't want one. The second line puts a margin of 10 pixels on the `li` elements to help give a little visual room between each list item. The third and fourth lines set a left margin on the `input` and `textarea` elements so that they don't crowd the labels and line up properly. If you would like more information on the styling of a form please consult the article on Styling Forms in this Web Standards Curriculum (to be published at a later date) or [Nick Rigby's A List Apart article on "Prettier Accessible Forms"](#).

Summary

In this article, I have covered the most basic three steps to a web standards compliant form. There is much more in form world I did not cover here and that is left for you to explore for now. There are access keys, checkboxes and radio buttons, custom submit / reset buttons, and select boxes.

In the above Step three Form, I added a checkbox to show how you can use the additional attributes in the `input` element to collect information that is beyond the single line text input or the multiple line text area input. The `checkbox` and `radio` button attribute values could be used to add the ability to ask short questions and give the reader a list of options to choose from (checkboxes allow you to select multiple options, radio buttons only one). Radio buttons can be very useful in a survey form.

The `select` element, also not featured in this article, can be used for a drop down menu of choices (for example a list of countries, or states/provinces).

Further reading

- Cameron Adams, “Accessible, stylish form layout”: <http://www.themaninblue.com/writing/perspective/2004/03/24/>.
- Brian Crescimanno, “Sensible Forms: A Form Usability Checklist”: <http://www.alistapart.com/articles/sensibleforms/>.
- Aaron Gustafson, “Learning to Love Forms”: <http://www.easy-reader.net/archives/2007/05/04/webvisions-wrapped/>.
- Simon Willison, “Easier form validation with PHP”, <http://simonwillison.net/2003/Jun/17/theHolyGrail/>.
- The Spec. Not any old specification, but THE W3C SPEC—<http://www.w3.org/TR/html401/interact/forms.html>. If you ever have a bout of insomnia in which a warm glass of milk, counting sheep, and [Ambien](#) are not putting you to sleep, go read the whole spec for HTML 4.01 or XHTML 1.0 at the w3.org. It is cheaper and more effective than any remedy out there. [Insert name of deity here] bless the engineers of the world.
- The nice folk over at the W3.org have defined the differences between “GET” & “POST” and when to use them: <http://www.w3.org/2001/tag/doc/whenToUseGet.html>.
- Programmers can be bribed with odd, neon colored snack foods.
- Many blessings upon Mr. Rigby: <http://alistapart.com/articles/prettyaccessibleforms>.

Exercise questions

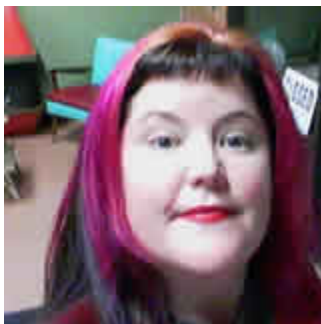
Time to code your own contact form.

1. Create a simple contact form that asks the user for their name, email address, and a comment.
2. Add a checkbox asking if the reader would like to join your mailing list.
3. Use some CSS to style your form: set a width to the form, align the labels to the left, put a background color to you page, etc.

Extra Credit: The more you play with the form elements and different CSS the more you will see what you can do with a simple contact form.

Extra Extra Credit: If you want to proceed into deep unknown lands, find a script or use one that your web host provides to send the form to yourself. If the script part to make the form gets a bit ornery, bribe a nice programming sort with neon snacks.

About the author



A web designer/developer by trade, a photographer, moblogger and professional art weirdo for the love of it, Ms. Jen started her art and design career in the high chair with her love of personal food art and food wall art at the age of 11 months. She taught herself HTML in 1996 when a computer snob said that an artist could not learn to code and has been fully in love with all things web design ever since.

Ms. Jen is the owner and founder of Black Phoebe Designs, a web & mobile design firm. Ms. Jen has has a Masters degree in Computer Science and Multimedia Systems from Trinity College in Dublin, Ireland, and taught web design at an LA area university from 2001-2005. She has participated in two Nokia mobile blogging projects, Wasabi Lifeblog (2004-2005) and the Nokia Urbanista Diaries (2008). Ms. Jen can always be found online at blackphoebe.com or blackphoebe.mobi.

21: Lesser-known semantic elements

BY [MNFRANCIS](#) · 8 JUL, 2008

Introduction

In this article I will introduce you to some of the more obscure and less well-known and used semantic elements in HTML. I'll look at marking up programming code, interaction with computers, citations and abbreviations, showing changes made to documents and more. I will also finish up by looking at some of the proposals for new extra semantics made in the draft of HTML 5.

- [Highlighting contact information](#)
- [Programming languages and code](#)
- [Displaying computer interaction](#)
- [Variables](#)
- [Citations](#)
- [Abbreviations](#)
- [Defining instances](#)
- [Superscript and subscript](#)
- [Line breaks](#)
- [Horizontal rules](#)
- [Changes to documents \(inserting and deleting\)](#)
- [Some future HTML elements](#)
- [Summary](#)

Note: After each code example, there is a “View source” link, which when clicked will take you to the actual rendered output of that source code, contained within a different file—it is provided so you can view live examples of how the source code is actually rendered in the browser, as well as looking at the code.

Highlighting contact information

The `address` element is probably the most badly named and misunderstood element in HTML. At first glance, with a name like “address” it would appear that it is used to encapsulate addresses, email, postal or otherwise. This is only partially the case.

The actual meaning of `address` is to supply contact information *for the author(s)* of the page, or the major section of the page, that it appears within. This can take the form of a name, an email address, a postal address or a link to another page with more contact information. For example:

```
<address>
  <span>Mark Norman Francis</span>,
  <span class="tel">1-800-555-4865</span>
</address>
```

[View source](#)

In the following example, the address is contained within the footer paragraph and simply links to another page on the site. The extended contact information on the page that this link targets could then have much more detailed contact information, to save repeating it endlessly across the entire site.

```
<p class="footer">© Copyright 2008</p>
<address>
  <a href="/contact/">Mark Norman Francis</a>
</address>
```

[View source](#)

Of course, if the site had more than one author, the same pattern could be used, just linking to different contact pages for the different authors.

It is *incorrect* to use the `address` element to indicate any other type of addresses, such as this:

```
<p> Our company address: </p>
<address>
  Opera Software ASA,
  Waldemar Thranes gate 98,
  NO-0175 OSLO,
  NORWAY
</address>
```

[View source](#)

(Of course, if Opera was taking collective responsibility for this article, this would be correct, even though I, and not all of Opera, am the author of this particular page.)

For any general address, you can use something called a "microformat" to indicate that a paragraph contains an address. There is [more information on Microformats in other articles on dev.opera.com](#).

Programming languages and code

The `code` element is used to indicate computer code or programming languages, such as PHP, JavaScript, CSS, XML and so on. For short samples within a sentence, you would simply wrap the element around the code snippet, like so:

```
<p>It is bad form to use event handlers like
<code>onclick</code> directly in the HTML.</p>
```

[View source](#)

For larger samples of code which span multiple lines, you can use a preformatted block as shown in the [Marking up textual content in HTML](#) article.

Although there is no defined method of indicating which programming language or code is shown within the `code` element, you can use the `class` attribute. A common practice (mentioned in the HTML 5 draft) is to use the prefix `language-` and then append the language name. So the above example would become:

```
<p>It is bad form to use event handlers like
<code class="language-javascript">onclick</code>
directly in the HTML.</p>
```

[View source](#)

Some programming languages have names that cannot be easily represented in classes, such as C# (C-Sharp). The correct way of writing this would be `class='language-c\#'`, which could be confusing and easily mis-typed. I would recommend using a class consisting of only letters and hyphens, and spelling it out completely. So in this case, use `class='language-csharp'` instead.

Displaying computer interaction

The two elements `samp` and `kbd` can be used to indicate the input and output of interaction with a computer program. For example:

```
<p>One common method of determining if a computer is connected
to the internet is to use the computer program <code>ping</code>
to see if a computer likely to be running is reachable.</p>
```

```
<pre><samp class="prompt">kittaghy%</samp> <kbd>ping -o google.com</kbd>
<samp>PING google.com (64.233.187.99): 56 data bytes
64 bytes from 64.233.187.99: icmp seq=0 ttl=242 time=108.995 ms

--- google.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 108.995/108.995/108.995/0.000 ms
</samp></pre>
```

[View source](#)

The `samp` element indicates sample output from a computer program. As shown in the example, different types of output can be indicated using the `class` attribute. There are no widely adopted conventions for which kind of classes to use, however.

The `kbd` element indicates input from the user interacting with the computer. Although this is traditionally keyboard input (hence the “kbd” contraction used) it should also be used to indicate other types of input, such as spoken voice.

Variables

The `var` element is used to indicate variables in textual content. This can include algebraic mathematical expressions or within programming code. For example:

```
<p>The value of <var>x</var> in
3<var>x</var>+2=14 is 4.</p>

<pre><code class="language-perl">
my <var>$welcome</var> = "Hello world!";
</code></pre>
```

[View source](#)

Citations

The `cite` element is used to indicate where the nearby content comes from—when quoting a person, a book or other publication, or generally referring people to another source, that source should be wrapped in a `cite` element. For example:

```
<p>The saying <q>Everything should be made as simple as possible,
but not simpler</q> is often attributed to <cite>Albert
Einstein</cite>, but it actually a paraphrasing of a quote which
is much less easy to understand.</p>
```

[View source](#)

Abbreviations

The `abbr` and `acronym` elements are used to indicate where abbreviations occur, and provide a method for expanding upon them without unnecessarily interrupting the flow of the document.

The text that is the abbreviation gets wrapped in the `abbr` element, and the full version is placed in the `title` attribute, like so:

```
<p>Styling is added to
<abbr title="Hypertext Markup Language">HTML</abbr> documents
using <abbr title="Cascading Style Sheets">CSS</abbr>.</p>
```

[View source](#)

An acronym is a type of abbreviation, with the difference that the result is accepted to be, and spoken as if it were, an actual word. An example is scuba, which is formed from the phrase “self-contained underwater breathing apparatus”. Whilst the HTML 4.01 specification allows for both `abbr` and `acronym` elements, there is some trouble trying to do the right thing here...

Internet Explorer (before version 8) doesn't recognise the `abbr` element, but does recognise `acronym`. Unfortunately, acronyms are a subset of abbreviations and it is incorrect to markup something like “HTML” using the `acronym` element.

Also, in the draft of HTML 5, the `acronym` element has been dropped in favour of standardising on `abbr` for both, as any acronym is also a valid abbreviation.

The best thing to do is to avoid using `acronym` and just stick to using `abbr` throughout your code. If you need to apply some visual styling to the `abbr`, you can place a `span` inside it and target that instead of the `abbr` so that all browsers will get the visual styles applied. More details will appear in a later article on “styling text”.

Defining instances

There is some confusion over the proper use of `dfn`, which is described in the HTML specification as “the defining instance of the enclosed term”. This is remarkably close to the idea of the `dl` element (definition term) used in definition lists.

The difference is that the term used in `dfn` does not have to be a part of a list of terms and descriptions and can instead be used as part of the normal flow of text, even in conversational style prose. Its use is recommended when an abbreviation is used more than once.

So, for example, in the article [The basics of HTML](#) earlier in this series, the abbreviation HTML appeared over forty times. To use the code “`<abbr title="HyperText Markup Language">HTML</abbr>`” each and every time it is used would be a waste of bandwidth, visually distracting and for screen reader users probably quite tiresome as HTML is expanded over and over, even though they would already have been told what it stands for. Instead, the code could be inserted for the point where it is first defined for the readers:

```
<p><dfn><abbr>HTML</abbr></dfn> ("HyperText Markup Language") is  
a language to describe the contents of web documents. [...]</p>
```

[View source](#)

Then later, whenever HTML is used, it can be marked up simply as “`<abbr>HTML</abbr>`”. A user agent could then make available to the user some method of retrieving the defining instance of that abbreviation. Unfortunately, no user agent currently does this, including screen readers. It would be better, then, to use the `title` attribute:

```
<p><dfn><abbr title="HyperText Markup Language">HTML</abbr></dfn>  
("HyperText Markup Language") is a language to describe the contents of web  
documents. [...]</p>
```

[View source](#)

Unfortunately, we have now doubled up on the expanded term for HTML which can be a problem for screen reader users. However, leaving out the visible expansion makes the document less useful for sighted users which will be the greater proportion of users in almost every case.

I would suggest that this is an acceptable trade-off when there are only one or two items requiring a definition. If you are very concerned about this, the code could instead appear as:


```
<p><abbr title="HyperText Markup Language">HTML</abbr>
(<dfn>HyperText Markup Language</dfn>) is a language to
describe the contents of web documents. [...]</p>
```

[View source](#)

However, the user agent then has to make an assumption when it sees a `dfn` alone as to what that applies to. Currently, no user agent actually does anything useful with `dfn`, so this may be acceptable to you.

In pages that require a larger number of definitions, it might be better to create a glossary section or page where the more rigorous definition list markup can be used.

This is an unfortunate instance where the specification has been created without clear guidelines on how an element is supposed to be used, and probably was not based upon any real-world usage of that element – otherwise there would be a method of combining the term with its full description or definition. The HTML 5 specification goes into a lot more detail about how `dfn` is to be used, but this is still in draft and not suitable for use on the web yet.

Superscript and subscript

To mark up a part of some text as being super- or subscripted (slightly raised or lowered compared to the rest of the text) you use the `sup` and `sub` elements.

Some languages require these elements for correct usage of abbreviations and it can be used when a small amount of mathematical content is being marked up, without resorting to using MathML (a specific, rather heavyweight mathematical markup language, created for the sole purpose of marking up heavyweight mathematical formulae).

An example of both types:

```
<p>The chemical formula for water is H<sub>2</sub>O, and it
is also known as hydrogen hydroxide.</p>
<p>The famous formula for mass-energy equivalence as derived
by Albert Einstein is E=mc<sup>2</sup> – energy
is equal to the mass multiplied by the speed of light
squared.</p>
```

[View source](#)

Line breaks

Because of the way HTML defines white space, it is not possible to control where lines of text break (such as marking up a postal address as a paragraph, but wanting the visual appearance to have each part of the address appear on a separate line) by simply pressing the Return key whilst writing the text.

A line break can be introduced into the document using the `br` element. However, this should only be used to force line breaks where they are required, and never to apply more vertical spacing between paragraphs or such in a document—that is more properly done with CSS.

Sometimes it might be easier to use the preformatted text block rather than inserting `br` elements. Or, if one particular part of some text is desired to be on a line by itself, but this is just a styling issue, it can be surrounded by a `span` element and set to display as a block level element.

So for example you could write the Opera contact address seen earlier in this article when talking about the `address` element like this instead:

```
<p>Our company address: </p>
<address>
```

```
Opera Software ASA,<br>Waldemar Thranes gate 98,<br>NO-0175 OSLO,<br>NORWAY</address>
```

[View source](#)

Of course, if you are writing XHTML rather than HTML, the element should be self-closing, like so: `
`.

Horizontal rules

A horizontal rule is created in HTML with the `hr` element. It inserts into the document a line, which is described to represent a boundary between different sections of a document.

Whilst some argue that this is inherently non-semantic and purely a visual, presentational effect, there is actually some precedent in literature for such an element to exist. Within a chapter (which could be described as a section within a book), a horizontal rule will appear between scenes that occur in different times and/or places. Also, poetry can use decorative breaks to separate different stanzas of the poem.

Neither use would justify the existence of a new header element, which is the accepted way of marking the boundaries between document sections.

The `hr` element has no uncommon attributes and should be styled using CSS if the default appearance is unsatisfactory.

Also, like the line break, if you are writing XHTML and not HTML, use the self-closing form—`<hr />`.

Changes to documents (inserting and deleting)

If a document has been changed since the first time it was available, you can mark these changes so that return visitors or automated processes can tell what has changed, and when.

New text (insertions) should be surrounded by the `ins` element. Text that has been removed (deletions) should be surrounded by the `del` element. If text has been changed, that is actually a deletion and insertion at the same point in the document; good form suggests having the deleted text first, followed by the insertion.

Both elements can take two attributes that give more meaning to the edits.

If the reason for the change is stated in the page or elsewhere on the web, you should link to that document or fragment in the `cite` attribute. This effectively says “This change happened because of this reason.”

You can also indicate the time at which the change was made by using a `datetime` attribute. The value should be an ISO-standard timestamp, which is generally of the form “YYYY-MM-DD HH:MM:SS ±HH:MM” ([more information is available on wikipedia](#)).

An example using both attributes:

```
<p>We should only solve problems that actually arise. As
  <cite><del datetime="2008-03-25 18:26:55 Z"
    cite="/changes.html#revision-4">Donald Knuth</del><ins
      datetime="2008-03-25 18:26:55 Z"
      cite="/changes.html#revision-4">C. A. R. Hoare</ins></cite>
  said: <q>premature optimization is the root of all
  evil</q>.</p>
```

[View source](#)

Some future HTML elements

As has been noted several times in this and some other articles, [HTML version 5](#) is being drafted at the moment. This will be the most radical update to HTML since its inception. By actually studying the patterns of HTML being used right now on the internet, rather than thinking about what might be useful to people, it stands a good chance of taking document semantics that are currently little more than convention and inserting them directly into the specification.

Some example elements slated to be introduced in HTML that could really improve the way we encode and use documents include:

- `header`—contains the header (masthead) of a page; normally consisting of a logo and title, maybe a short “about” area and some site-global navigation such as login/logout/profile links.
- `footer`—contains the footer of a page, which normally consists of further links within a site, copyright and other legal information.
- `nav`—contains the primary navigation links of a page.
- `article`—contains the part of a page that is the main content area, excluding all other page elements such as navigation, header and footer.
- `aside`—contains sidebar information on a given area of the page, and can also be used for pull quotes or notes within the main content.

There are more, which you can find in the [HTML 5 specification](#) itself.

Summary

In this article, I have described some of the lesser known and more infrequently used semantic elements available in HTML. In the next article, available soon, we will examine further how to correctly use the two semantically-neutral elements in HTML, `div` and `span`.

About the author



Photo credit: [Andy Budd](#).

Mark Norman Francis has been working with the internet since before the web was invented. He currently works at Yahoo! as a Front End Architect for the world's biggest website, defining best practices, coding standards and quality in web development internationally.

Previous to Yahoo! he worked at Formula One Management, Purple Interactive and City University in various roles including web development, backend CGI programming and systems architecture. He pretends to blog at <http://marknormanfrancis.com/>.

22: Generic containers—the div and span elements

BY MNFRANCIS · 26 SEP, 2008

Published in: [DIV](#), [SPAN](#), [GROUPING](#), [HOOKS](#), [SEMANTIC](#)

This is Article 22 of the Web Standards Curriculum.

[Previous article—Lesser known semantic elements](#)

[Next article—Creating multiple pages with navigation menus](#)

[Table of contents](#)

Introduction

In this article, I am going to explain to you how and when to use the two elements in HTML that are *not* used to describe the content. The `span` and `div` elements do not actually confer any meaning to the content they envelop; instead, they are a generic mechanism that allows you to create custom structure or groupings of elements where no other HTML element is really appropriate, and that can be then styled with CSS or manipulated via JavaScript. Although `div`s don't add any semantic meaning, they could be considered to represent a structural division of the mark-up, along with the appropriate semantic class or ID name.

They are the “tag of the last resort” and should only be used where no other HTML element fits the bill, because they have no meaning to assistive technologies, search engines, etc.

The article structure is as follows:

- [Semantically neutral](#)
- [Inline versus block](#)
- [More explanation of grouping content](#)
- [Extra information](#)
- [Hooks for JavaScript, as well as CSS](#)
- [div-it-is](#)
- [Inappropriate semantics](#)
 - [Generic paragraphs](#)
 - [Presentational elements](#)
- [Summary](#)
- [Exercise questions](#)

Semantically neutral

Most elements in HTML exist to describe the content, such as images, lists, headings, or assist in setting up the document—head, body, link, meta, etc. There are two elements however that have no assigned meaning. From the HTML spec:

The `div` and `span` elements, in conjunction with the `id` and `class` attributes, offer a generic mechanism for adding structure to documents.

These two elements can be considered the scaffolding of HTML. They give you the ability to group content, add extra information around content and hooks for adding styling and interactivity. They do not however add any new semantic meaning to the document, in and of themselves.

Inline versus block

As you learned earlier, [block elements are elements that help inform the structure of a document](#). The `div` element, short for division, is the block level generic container. It is normally used to wrap around other

other block level elements, to group them together (see the next section for more of an exploration of this). It can also be used to collect together a bunch of inline elements and/or text that otherwise don't logically fit under another block level element, but this should be a last resort.

The `span` element is the inline level generic container. It also helps to inform the structure of document, but it is used to group or wrap other inline elements and/or text, rather than block level elements.

The line between the two different types might seem fairly arbitrary at first. The difference to bear in mind is the type of content, and how it would appear when written down without any styling. A `div` is placed around a group of block level elements—for example, to wrap a heading plus a list of links to make a navigation menu. A `span` wraps a group of inline elements or (most usually) plain text. The key word is “group”: if a `div` wraps just one block-level element, or a `span` just one inline element, it's being used unnecessarily. For example, check out the way the `div` and `span` elements are used in the following simple markup:

```
<body>
  <div id="mainContent">
    <h1>Title of the page</h1>
    <p>This is the first paragraph of content on my example page.</p>
    
    <p>This is the second paragraph of content on my example page. It is very
similar to the first, but there is a <span id="specialAlert">special alert
here
that we want to colour and increase text size using CSS</span>.
It's not really standard emphasis - it's more just styling, so <em> and
<strong> are not really appropriate.</p>
  </div>
</body>
```

You could now select the content inside the `div` and `span` using their `id` attributes, and apply special styling and positioning to them using CSS.

More exploration of grouping content

Viewing the source of many pages on the internet will reveal a nest of `div` elements including common metaphors in the `classes` and/or `ids` of the elements—for example `header`, `footer`, `content`, `sidebar`, and so on.

Your `class` and `id` names should always be *semantic*, meaning they should refer to the meaning/role of the content, rather than just referring to its visual appearance. So for example, `sidebar` and `alertMessage` are good `class` names, whereas `redLefthandColumn` and `blueFlashingText` are not. What if you wanted to change the colour of the sidebar from red to blue at a later date, or switch its position on the site from left to right? What if you wanted your alerts to be changed from blue and flashing to green and not flashing?

These divisions provide predictability when creating page structures, and, perhaps most importantly, when looking at the HTML again later, they provide clues as to which part of the page you are in. A well divided page is almost self documenting as to its intent and contents.

To hopefully make this a little bit clearer, let's look at a `div` structure from a real site—[the home page of dev.opera.com](http://dev.opera.com) to be exact. Bear in mind that the below code example contains no content at all, apart from a few other elements I've included because they are important for the site structure. I'm mainly looking at reproducing just the actual site structure as defined using `div` elements. In the code below, read the HTML comments carefully—I've inserted these to explain the site structure. As you look through the code, open the main page of dev.opera.com inside a new tab or window in your browser so you can refer to the look of the site as you explore its structure.

```
<body>
<!-- First up we have a wrap div, which wraps the entire page, and allows more
precise control of it as a whole -->
  <div id="wrap">
    <!-- This unordered list contains the list of links to all Opera's different
    sites, which you can see across the very top of the page -->
      <ul id="sitenav" class="hidemobile">
        ...
      </ul>
      ...
    <!-- This is the search form - the search box you see at the top right of the
    page -->
    <form action="/search/" method="get" id="search">
      <div>
        ...
      </div>
    </form>
    <!-- This unordered list contains the main navigation menu of the site - the
    horizontal tab menu you see just below the main title graphic -->
    <ul id="menu">
      ...
    </ul>
    <!-- This nested div forms the structure of the login box, where you enter
    your user name and password to log in to the site. You will only see this if you
    are currently logged out. -->
    <div id="loginbox">
      <div id="login">
        ...
      </div>
    </div>
    <!-- This series of nested divs is where the main content of the page is
    contained - all of the article summaries that form the main bulk of the page
    content -->
    <div id="content2">
      <div id="main">
        ...
        <div class="major">
          ...
        </div>
        <div class="major">
          ...
        </div>
      </div>
    </div>
    <!-- This div contains the sidebar of the page - the article categories, the
    latest comments, etc -->
    <div id="side">
      ...
    </div>
    <!-- This div contains the page footer, which is where you'll see the
    copyright message, and various links at the bottom of the page. -->
    <div id="footer">
      ...
    </div>
    <!-- The end of the page - this is the closing tag for the wrap div -->
  </div>
</body>
```

Extra information

Some content has extra information that is of use to user agents and other parsers, and this needs to be conveyed through an attribute. `span` elements are often a good way to attach such information to content on a web page, as you will see below.

A good example is a different language appearing within a document. For example:

```
<p><q>Plus ça change, plus c'est la même chose</q> she said.</p>
```

Although the language of the main document is English, the quote is actually French. This would be indicated through the use of the `lang` attribute, like so:

```
<p><q lang='fr'>Plus ça change, plus c'est la même chose</q> she said.</p>
```

Now, in that example, it was easy to mark the change in language as it all appeared within a quote, so the `q` element was perfect to use to wrap the content. There are some cases however where there isn't an appropriate semantic element easily available, so you would instead have to resort to a `span` or `div`. For example:

```
<p>A screen reader will read the French word chat (cat) as chat (to talk informally) unless it is properly marked up.</p>
```

In this example, the first instance of the word *chat*, being an example of French within an English document, should have the difference indicated so it isn't just interpreted as the English word chat. In this case, a `span` around the word chat is the right way to go about it, as there is no other HTML element appropriate to wrap the French word (we do not want to emphasise the word, it is not a quote, or a piece of code, etc). And being a single word in a sentence, it is inline level. The example would therefore be better written as:

```
<p>A screen reader will read the French word <span lang='fr'>chat</span> (cat) as chat (to talk informally) unless it is properly marked up.</p>
```

This is the same technique used in [microformats](#) for marking up common data formats within web pages. You can find a lot more about microformats in some of the [more advanced HTML articles on dev.opera.com](#).

Hooks for JavaScript, as well as CSS

I've already talked about how you can use `div` and `span` along with `id` and `class` attributes to provide hooks with which to apply CSS styles and positioning to certain parts of your content. The same thing can be done to apply JavaScript to your document too.

If a given element needs to be found and manipulated by JavaScript, it is common to apply an `id` to it, and then use the `getElementById` function to find it. You'll learn a lot more about JavaScript in the last part of the course.

"div-itis"

One thing to be aware of is an effect commonly referred to as "div-itis" in the web development community.

Whilst it is easy to add styling via a lot of nested `div` or `span` elements, it is a temptation best avoided as much as possible. In most cases, it is possible to attach the styling or JavaScript functionality to existing elements in the document. A generic container should be a last resort—it is better to try to write web pages by starting with just the semantic elements, and adding the containers only when necessary.

Inappropriate semantics

In this section I explore some common mistakes to be aware of when marking up content using HTML, to be avoided if at all possible.

Generic “paragraphs”

Sometimes, it is tempting to throw a `p` (paragraph) element around any block of text, but this isn’t really correct. As stated in my earlier [article on marking up content](#):

If it is just a few words and not even a proper sentence, then it should probably not be marked up as a paragraph.

A `div` or `span` (depending on the exact situation) is the correct element to wrap around disjointed textual content that has no other semantic relationship covered by other HTML elements.

Presentational elements

One notably bad piece of advice sometimes found on the internet is the practice of using short, presentational elements such as `b` or `i` as generic containers in place of `span`. The reasoning provided is commonly one of two things:

- The elements are three bytes shorter, and so save bandwidth in both the HTML and the CSS.
- The styling required is for appearance only, so using “presentational” elements is actually ok in this circumstance.

The first one is true, but the saving is almost always negligible (unless you are doing an incredible amount of presentational effects), especially given modern compression applied by web servers to documents before sending them over the internet to the browser. This makes documents much shorter than any human short-hand can achieve.

The second reason betrays a lack of comprehension as to what presentational actually means in the context of HTML. Presentational elements describe what their content should *look like* (`b` means simply “text within is bold”). They do not represent generic hooks for styling what is within.

If a small section of text within a paragraph needs styling or targeting by JavaScript and there is not already an applicable semantic element to wrap around it, the only correct element to use is a `span`.

Summary

This draws my exploration of the `span` and `div` elements to a close—you should now understand their purpose much better, and be able to use them with confidence. Later articles on CSS will go much more deeply into using them to create page layouts, and other uses.

Exercise questions

- What is the difference between `div` and `span`?
- Name 2 main uses of these elements on web pages.
- Have a look at the source code of one of the pages of your favourite web site. Consider the structure of it. Does it use a lot of `div` and `span` elements? Can you see anything bad or inappropriate about the way they are used? How could it be improved?

About the author



Photo credit: [Andy Budd](#).

Creative Commons Attribution, Non Commercial - Share Alike 2.5 license.
re ASA. All rights reserved.

Mark Norman Francis has been working with the internet since before the web was invented. He currently works at Yahoo! as a Front End Architect for the world's biggest website, defining best practices, coding standards and quality in web development internationally.

Previous to Yahoo! he worked at Formula One Management, Purple Interactive and City University in various roles including web development, backend CGI programming and systems architecture. He pretends to blog at <http://marknormanfrancis.com/>.

23: Creating multiple pages with navigation menus

BY [CHRISTIAN HEILMANN](#) · 26 SEP, 2008

Introduction

In this article I'll talk about web site navigation and menus. You'll learn about different types of menus and how to create them in HTML. I'll also touch on the subject of menu usability and accessibility. I won't go into styling menus yet, but this article will lay the foundations for a corresponding CSS tutorial later in the course. There are [code examples to download](#) to go along with this article—I will refer to these throughout the tutorial. The table of contents for this article is as follows:

- [Your HTML menu tools—links, anchors and lists](#)
- [The need for flexibility](#)
- [Different types of menu](#)
 - [In-page navigation \(table of contents\)](#)
 - [Site navigation](#)
 - [Providing visitors with a “You are here” feeling](#)
 - [How many options should you give users at one time?](#)
 - [Contextual menus](#)
 - [Sitemaps](#)
 - [Pagination](#)
- [When lists are not enough—image maps and forms](#)
 - [Setting hotspots with image maps](#)
 - [Saving screen space and preventing link overload with forms](#)
- [Where to put the menu, and offering options to skip it](#)
- [Summary](#)
- [Exercise questions](#)

Your HTML menu tools—links, anchors and lists

There are several different types of menu and navigation idioms to consider in HTML, all connected closely with `link` and `a` (anchor) elements. In a nutshell:

- `link` elements describe relationships across several documents. You can for example tell a user agent that the current document is part of a larger course that spans several documents and what other document is the table of contents.
- Anchors (aka `a` elements) allow you to either link to another document or to a certain section of the current document. These don't get automatically followed by the user agent; instead they'll be activated by your visitors by whatever mean available to them (mouse, keyboard, voice recognition, switch access...)

If you haven't read the [links article](#) and [lists article](#) earlier in the course, I'd like you to go back and have a read as I build on some of the information given there to avoid repetition.

Anchors/links do not just become menus on their own—you need to structure and style them to let both the browser and your users know that their function is as a navigation menu, not just a set of random links. If the order of the pages is not important you can use an unordered list as in this [unordered list menu example](#).

Note that I've put an `id` on the `ul` elements. The reason for that is to provide a hook for styling it with CSS and adding special behaviour with JavaScript later on. An `id` is a very inexpensive way to allow other technologies to single out a certain element in HTML.

If the order in which the visitors go through all the documents is important then you need to use an ordered list. If for example you have a multi-document online course where one tutorial builds on top of the last one, you could use an ordered list like in this [ordered list example](#).

Using lists to create menus is great for several reasons:

- All the HTML is contained in a single list element (`ul` for example), which means you can use the cascade in CSS to style each differently and it is easy to reach the elements in JavaScript going from the top level down.
- Lists can be nested, which means you can easily create several levels of navigation hierarchy.
- Even without any styling applied to the list, the browser rendering of the HTML makes sense and it is easy to grasp for a visitor that these links belong together and make up one logical unit.

You nest lists by embedding the nested list inside the `li` element, not after it. you can see a [correct and an incorrect example here](#).

Notice that browsers display both examples in the same way. Browser display should never be an indicator for the quality of your code. An invalid HTML construct like the wrong example shown on the above example page will be hard to style, add behaviour to with JavaScript or convert to another format. The structure of nested ULs should always be `` and never ``.

The need for flexibility

The menu of a site is unlikely to stay the same for very long—sites tend to grow organically as functionality is added and the user base grows, so you should create menus with scope for menu items to be added and removed as the site progresses, and for menus to be translated into different languages (so links will change in length). Also, you may well find yourself working on sites where the HTML for menus is created dynamically using server-side languages rather than with static HTML. This does not however mean that knowing HTML will become obsolete; it will actually become more important as this knowledge will still be needed to create HTML templates for the server-side script to work with.

Different types of menu

There are several types of menus you will be called upon to create in HTML, as you work on different web sites. Most of these can be created with lists, although sometimes interface restrictions force you to go another route (more on that later). The list-based menus you will be likely to create are as follows:

- In-page navigation: for example a table of contents for a single page, with links pointing to the different sections on the page.
- Site navigation: a navigation bar for your whole web site (or a subsection of it), with links pointing to different pages on the same site.
- Content-contextual navigation: a list of links that point to pages closely related to the subject of the page you're currently on, either on the same site, or different ones.
- Sitemaps: large lists of links that point to all the different pages of a web site, grouped into related sublists to make them easier to make sense of.
- Pagination: links pointing to other pages that make up further sections or parts of a whole, along with the current page, for example part 1, part 2, and part 3 of an article.

In-page navigation (table of contents)

I already covered this to a certain degree in the tutorial about links, but here's a more in-depth description of what in-page navigation means and what you need to make it work.

In the [example related to this in-page navigation section](#) I have used a list of links that point to anchors further down the page. In order to connect the anchors you use an `id` attribute on the elements that are to be navigated to and an `href` attribute consisting of a hash sign followed by the same name as the `id` value of the anchor you want this link to point to. Each section of the page also has a "back to menu" link that works in the same way, but points back to the menu instead.

Technically, this is all you need to make this kind of navigation work, however, there is an annoying bug in Internet Explorer that forces you do to a bit more.

You can try this bug out yourself:

1. Open the document in Internet Explorer 6 or 7
2. Do not use a mouse; instead use the keyboard to navigate the document. You can hit the tab key to jump from link to link and the enter key to activate a link—in this case to jump to the section it points to.
3. Seemingly all is well when you do that—the browser scrolls down to where you wanted to go.
4. If you hit the tab key again the right behaviour would be for a browser to take you to (give focus to) the first link inside the section you chose. Internet Explorer, however, will take you back to the start of the menu at the top of the page!

The way around this is terribly confusing and deals with a special property of Internet Explorer called `hasLayout`. You can trigger this in several ways, all of which are explained in [Ingo Chao's excellent article "On having layout"](#). The easiest way is to [set the width of the `div` element using CSS, as in this example](#).

This is what IE needs—the anchor to be inside an element with `hasLayout`.

Having to do this is annoying, but it also helps you if you want to style the sections differently. It also works around one of the problems of headings in HTML: headings don't contain the sections they apply to; it is just assumed that everything that follows until the next heading is part of the same document section. This makes it impossible to style different sections of a document without adding a `<div>`. Other markup languages propose a `<section>` element with a `<title>` inside it, much like `<fieldset>` and `<legend>`, which would allow you to mark up parts of a form.

Note that keyboard navigation around links in Opera is slightly different—try looking at the above example in Opera, then hold down `Shift` and use the arrow keys to navigate around links (it also works on form elements). This is called spatial navigation.

Site navigation

Site navigation is most probably the most common menu type you'll need to create. It is the menu of the whole site (or a subset of it), showing both the options visitors can choose from and the hierarchy of the site. Lists are perfect for this purpose, as you'll see from this [site navigation example](#).

There aren't many surprises here, at least not from a pure HTML point of view. Later on in the course we'll talk about styling these kind of menus with CSS and adding behaviour via JavaScript. One important thing to consider is how to highlight the current document in the menu, to give the user a sense of being in a particular place, and that they are moving location (even though in reality they aren't, unless of course they are using a mobile device to browse the Web!). This is what I'll look at next.

Providing visitors with a "You are here" feeling

One golden rule of web development and navigation is that the current document should never link to itself but instead be obviously different to the other entries in the menu. This is important as it gives the visitors something to hold on to and tells them where they are on their journey through your site. There are edge cases like web applications, permalinks in blogs and so called "one page" web sites but in 99% of cases a link to the document you are already looking at is redundant and confusing to the visitor.

In the links tutorial, I stated that a link is an agreement and a liability: you offer visitors a way to reach more information that they need, but you need to be careful—you'll lose credibility and trust if that link doesn't give the users what they want, and/or results in unexpected behaviour. If you offer for example a link that points to the current document, activating it will reload the document. As a user this is something you don't expect—what purpose did clicking this link have? This results in users getting confused.

This is the reason why the current page should never be linked to from the menu. You could remove it altogether, or even better stop it from being a link but highlight it (eg by surrounding it with a `strong` element)—this gives users a visual clue and also tells blind visitors that this is important and the current entry in the menu—this [current page highlight example](#) illustrates this.

How many options should you give users at one time?

Another issue to consider is how many options you want to give visitors. A lot of menus you see on the web try to make sure that every page in the site can be accessed from one single menu. This is where scripting and CSS trickery comes in—you can make the menu more manageable by hiding parts of the menu until the users select certain areas (rollover menus, as they are sometimes called). This is clever from a technical point of view, but there are several issues with this approach:

- Not all visitors will be able to use the clever trick as intended; keyboard users for example will have to tab through all links on the page just to reach the one they are looking for.
- You need to add a lot of HTML to each document of your site to achieve this, and a lot of it can be redundant on many pages. If I drill down three levels in your menu to reach a document I want to read, I don't need to see options leading me to 4, 5, and 6 levels deep.
- You can overwhelm visitors if you present them with too many options at once—humans don't like making decisions. Think about how long it can take you to pick a meal from a lengthy restaurant menu.
- If there is not much content on a page but a lot of links, search engines will assume that there is not much valid information on this page and not give the page much attention, so it is harder to find when searching the Web.

All in all, it is up to you how many items you put into a menu—different designs will call for different choices—but if in doubt, you should try cutting your menus down to only the links to the main sections of the site. You can always provide further submenus where appropriate.

Contextual menus

Contextual menus are links that build on the content of the current document and offer more information related to the current page you are on. A classic example is the “related articles” links you tend to get at the bottom of news articles, as shown in Figure 1.

Related Articles: Party Loans Investigation

- ▶ [Top minister denies new funding claims](#) AFP - 49 minutes ago
 - ▶ [Health secretary denies new funding claims](#) AFP - Sunday, January 27 04:47 pm
 - ▶ [Brown enjoying 'world's best job'](#) Press Assoc. - Sunday, January 27 12:17 pm
 - ▶ [Health secretary 'failed to declare donations'](#) AFP - Sunday, January 27 09:01 am
 - ▶ [PM denies 'dithering' over Hain](#) Press Assoc. - Friday, January 25 04:34 pm
-

Related Articles: Politics

- ▶ [Top minister denies new funding claims](#) AFP - 49 minutes ago
- ▶ [Johnson 'took third party donations'](#) Epolitix - Sunday, January 27 10:33 am
- ▶ [Rock collapse 'has hit UK competitiveness'](#) Epolitix - Sunday, January 27 03:18 am
- ▶ [Purnell to press private sector role](#) Epolitix - Sunday, January 27 02:09 am
- ▶ [Ashdown out of running for Afghan role](#) Epolitix - Sunday, January 27 01:56 am

Figure 1: An example of a contextual menu—a news article offering related news items at the bottom.

This is a slightly different thing to context menus in software user interfaces, which offer different options depending on where they are accessed (like the right-click or Ctrl + click menus you find in desktop applications that offer specific options depending on where your mouse pointer is at the time).

Contextual menus on web sites are a great way to promote content on other parts of the site; in terms of HTML they are just another list of links.

Sitemaps

Sitemaps are what you might expect—maps of all the different pages (or the main sections if you are talking about really huge sites) of your site. They allow your site's visitors to get a glimpse of the overall

structure of your site, and go anywhere they need to fast—even if the page they need is deep within your page hierarchy.

Both sitemaps and site searches are a great way of offering visitors a fallback option when they are lost or to offer quick access for those who are in a hurry.

From an HTML point of view they could either be one massive nested list full of links or—in the case of very large sites—section headings with nested links of section-specific hierarchies or even search forms for each of the sections.

Pagination

Pagination is necessary when you have to offer a way to navigate through large documents split into separate pages. You'll find pagination for example in large image archives or search result pages (like Google or Yahoo search.)

Pagination is different from other types of navigation because it does normally link back to the same document—but with a link that has more information in it like which page to start from. Some examples of pagination are shown in Figure 2:

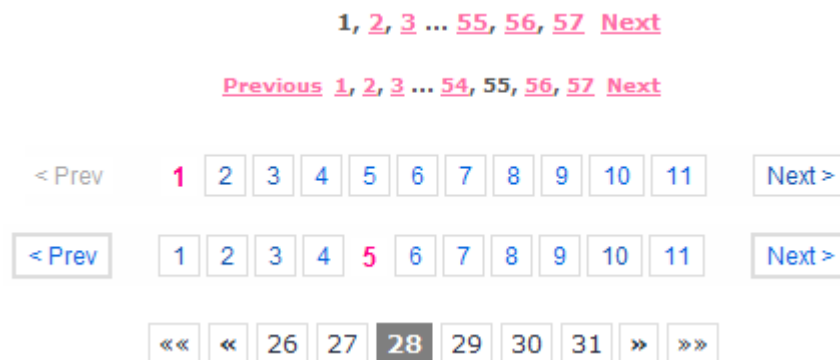


Figure 2: Pagination menus allow visitors to go through large sets of data without losing track of where they are.

The HTML is nothing ground-breaking—once again you offer a list of links with the current link (indicating which chunk of data is shown and how far down in your pagination you are) not being linked and highlighted (eg with a `strong` element).

The main difference to site navigation is that there is a lot of programming logic going on with pagination. Depending on where you are in the whole data set you need to show or hide the previous, next, first and last links. If you have really massive amounts of information to navigate through you also want to offer links to landmarks like 100, 200 and many more options. This is why you are not very likely to hard-code menus like these in HTML but instead create them on the server-side. This does not change the rules however—the current chunk should not link to itself and you shouldn't offer links that lead nowhere.

When lists are not enough—image maps and forms

In 99% of the cases the ordered or unordered list is a sufficient HTML construct for menus, especially as the logical order and nesting also allows for styling with CSS very nicely. There are however some situations that may require different design techniques.

Setting hotspots with image maps

One technique is client side image maps. Image maps turn an image into a menu by turning sections of the images into interactive areas that you can link to different documents. The [imagemap example](#) associated

with this section turns an image into a clickable menu. Try it out by following the link above and clicking the different sections of the triangle in the image shown in Figure 3.

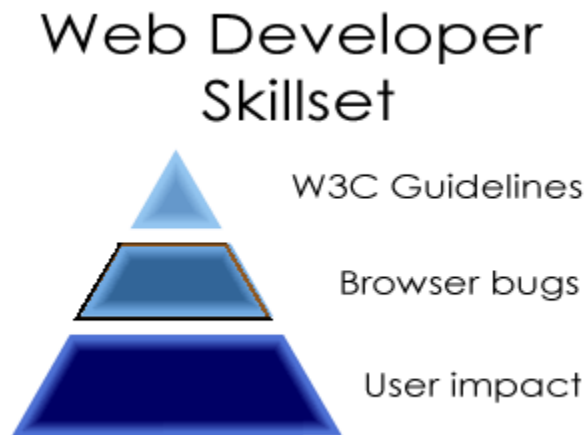


Figure 3: By defining a map with area elements you can turn parts of an image into interactive elements.

You can turn any image into a menu by defining a map with different areas (also called hotspots). You give the map a `name` attribute and connect the image and the map using the `usemap` attribute on the `img` element. Notice that this works exactly like in-page links, which means that you need to precede the value of the `usemap` attribute with a hash.

Each area should have several attributes:

`href`

defines the URL the area should link to (which could also be a target in the same document)

`alt`

defines alternative text in case the image can not be found or the user agent does not support images

`shape`

defines the shape of the area. This can be `rect` for rectangles, `circle` for circles or `poly` for irregular shapes defined using polygons.

`coords`

defines the coordinates in the image that should become hotspots—these values are measured from the top left corner of the image, and can be measured in pixels or percentages. For rectangular areas you only need to define the top left and the bottom right corner; for circles you need to define the center of the circle and the radius; for polygons you need to provide a list of all the corner points.

Image maps are not much fun to define and type in as HTML, which is why image manipulation tools like Adobe Image Ready or Fireworks offer an option to create them visually (they generate the HTML for you).

Saving screen space and preventing link overload with forms

Another technique you can employ is to create a form using a `select` element. You can define your different pages as options inside the `select` element, and your visitors can choose an option then submit the form to jump to different pages. You can find a working [form menu example here](#).

The most obvious benefit of using this type of menu is that you can offer a lot of options without using up a lot of space on the screen, as browsers render the menu as one line –see Figure 4.

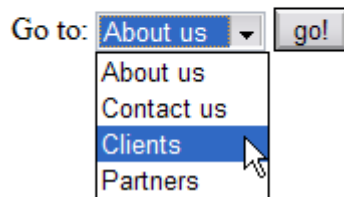
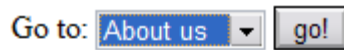


Figure 4: Select menus take up only one line on the screen.

You can go further with this, grouping the different menu options using the `optgroup` element, as shown in this [optgroup example](#).

This will show a menu with non-selectable options (these are the group names) as shown in Figure 5:

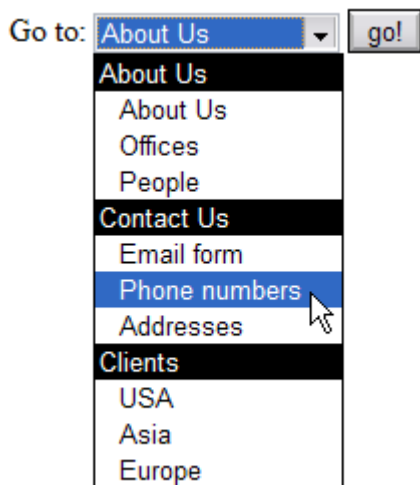


Figure 5: Select menus can get option groups that allow you to tell visitors which options belong together. This is the rendering on Opera 9.5.

This technique has the benefit of using up hardly any space but it also means that you need to have a server-side script to send the visitors to the pages they have chosen. You can also use JavaScript to make the links work, but you cannot rely on JavaScript being available—you need to make sure your users can still make use of the menu with JavaScript disabled.

Another, less obvious benefit is that you don't offer too many links in the same document. This means that you don't overwhelm users of assistive technologies (who often tend to be presented with the links in one big list). It also means that search engines don't consider the links on your page worthless as the link to text ratio makes the document appear to be a sitemap. However, many assistive technologies can produce

a map of your pages' links; if your important link are all in a select menu, there is a chance that a visitor might never chance upon them. It is therefore a good idea to offer anchor links to the main destination pages and `select` element menus to offer more options. Visitors will be able to use them, but machines like search engine robots don't need to know they exist.

Where to put the menu, and offering options to skip it

One last thing to mention about HTML menus is that the placement of the menu plays a large role. Consider visitors that have no scrolling mechanism or might not be able to see so rely on keyboard navigation to find their way around your site. The first thing they'll encounter when they load the document is its location and the title; next the document gets read top to bottom, stopping at each link to ask the visitor if they wanted to follow this link or not. Other options are to get a list of all the links or to jump from heading to heading.

If the menu is at the top of document, it will be the first thing the user will meet. Having to skip through 15 or 20 links before they get to any content could get really annoying. There are two workarounds available. First, you could put the menu after the main content of the document (you can still place it at the top the screen using CSS if you wish). Second, you could offer a skip link. Skip links are simply links placed before the main menu that link to the start of the content, allowing the visitor to skip over the menu and get to the content immediately if they wish. You can add another "go to menu" link at the end of the document to make it easy to get back up to the top. Check out the [skiplinks example for more of an insight](#).

Skip links are not only useful for these kind of disabilities but make life a lot easier when you navigate a site on a mobile device with a small screen.

Summary

In this tutorial I covered the different types of menu you are likely to have to write in HTML. I've talked about:

- Why lists with anchors are the perfect HTML construct to define a menu
- Why it is important to not consider menus as set in stone but to expect and plan for change instead
- In-page navigation: linking to sections of the current document and back to the menu
- Site navigation: offering a menu that shows both the pages in the current site and their hierarchy; I also looked at why it is important to highlight the page the user is currently looking at
- Contextual navigation: offering links to related pages elsewhere on the site (or on other sites)
- Sitemaps: as a fallback and re-orientation tool for visitors that feel lost or come with a specific need
- Pagination: a tool to allow visitors to navigate through a document split up into multiple pages
- Image maps: creating graphical menus by overlaying images with hotspots
- Form menus: providing a lot of options without using up a lot of space and without overwhelming your visitors and search engines with too many links.
- Skip links and menu placement

We will come back to some of these topics later on in the CSS section of this course to take a look at how to make these HTML constructs look good and become even more obvious as a menu for your visitors.

Exercise Questions

- Why is it a good idea to mark up menus as lists?
- When you design a navigation menu, what do you need to plan for in the future?
- What are the benefits of using `select` elements for menus, and what are the problems?
- What do you define with `area` elements, and what is the `nohref` attribute of an area element for (this is not in here, you'd need to do some online research)
- What are the benefits of skip links?

About the author

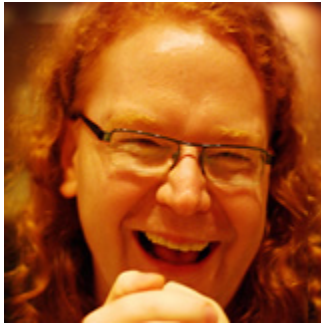


Photo credit: [Bluesmoon](#)

Chris Heilmann has been a web developer for ten years, after dabbling in radio journalism. He works for Yahoo! in the UK as trainer and lead developer, and oversees the code quality on the front end for Europe and Asia.

Chris blogs at [Wait till I come](#) and is available on many a social network as “codepo8”.

24: Validating your HTML

Introduction

So you've written a few HTML pages, and they seem to display ok to you, but there are a few things not quite right with them. What is the best way to start finding out what is wrong, and ensure that these pages (and any future pages you write) will be displayed properly across browsers, with no errors?

Validation is the answer! There are many tools available, from the W3C and other places, that allow you to validate the code on your sites. The most common three validators you'll use are:

- [The W3C Markup Validator](#): This looks at the (X)HTML doctype you are using for the document you give it to check, and goes through the entire document pointing out places where your HTML doesn't follow that doctype correctly (ie where there are errors in your HTML).
- [The W3C Link Checker](#): This looks through the document you give it to check, and tests all the links inside that document to make sure they are not broken (ie the `href` values point to resources that don't exist).
- [The W3C CSS Validator](#): As you've probably guessed, this will go through a CSS (or HTML/CSS) document and check that the CSS follows the CSS specs properly.

In this article, I will cover the first of these, showing you how to use it, and how to interpret the typical kinds of results the validator gives you. The link checker is very obvious, and the CSS validator should be fairly self explanatory after you've read this article, and the CSS articles that come later on in the course.

The structure of this article is as follows:

- [Errors](#)
- [What is validation?](#)
- [Why validate?](#)
- [Different browsers interpret invalid HTML differently](#)
 - [Quirksmode](#)
- [How to validate your pages](#)
 - [The W3C HTML validator](#)
- [Summary](#)
- [Further tools to check out](#)
- [Exercise questions](#)

Errors

In computer programming, there are broadly speaking two kinds of problems with code:

- syntactical errors—these are where a mistake in the writing of the code causes the computer to be unable to execute or compile the program properly.
- programming (or logic) errors—these are where the code does not completely reflect the intent of the programmer.

With most programming languages, the first error is incredibly easy to spot—your program will just refuse to run or compile until the error is fixed. This makes finding and fixing these types of bugs much easier in those general head-scratching moments of “So why isn't it doing what I want?”

HTML is not a programming language. Syntax errors in a web page do not commonly cause the web browser to refuse to open the page (although XHTML tends to be a lot stricter than HTML—at least when served as `application/xhtml+xml` or `text/xml` data, as intended—and some doctypes will disallow certain types of HTML element from being used). This is one of the biggest reasons for the rapid adoption and spread of the web.

[The first web browser, WorldWideWeb](#) (written by Tim Berners-Lee) was also an editor, allowing people to create web pages without learning HTML first. This editor created invalid HTML. This could have been

fixed, but it established an important precedent that exists in all web browsers to this day—that allowing people access to the content is more important than complaining about errors to people that won't understand them or be in a position to fix them.

What is validation?

Although web browsers will accept bad (*invalid* is the term we use) web pages and do their best to render the code by making a best guess of the author's intention, it is still possible to check whether the HTML has been written correctly, and indeed it is a good idea to do so, as you'll see below. We call this "validating" the HTML.

The validation program compares the HTML code in the web page with the rules of the accompanying [doctype](#) and tells you if and where those rules have been broken.

Why validate?

There is a common feeling amongst some web developers that if a web page looks fine in browsers, it doesn't matter if it doesn't validate. They describe validation as an ideal goal, but not something that is a black-and-white issue.

There is some wisdom in this attitude. The HTML specification is not perfect, and it is now quite out of date. Some things that are arguably correct (such as [starting an ordered list at a point other than 1](#)) are invalid HTML. However, as the saying goes:

Learn the rules so you know how to break them properly.

There are two over-ridingly powerful reasons to validate your HTML as you author it.

- You are not always perfect, and neither is your code—we all make mistakes, and your web pages will be higher quality (ie, work more consistently) if you weed out all the mistakes.
- Browsers change. In the future, it is likely that browsers will be less forgiving when parsing invalid code, not more forgiving.

Validation is your early-warning system about introducing bugs into your pages that can manifest in interesting and hard-to-determine ways. When a browser encounters invalid HTML, it has to take an educated guess as to what you meant to do—and different browsers can come up with different answers.

Different browsers interpret invalid HTML differently

Valid HTML is the only contract you have with the browser manufacturers. The HTML specification says how you should write it, and how they should interpret your document. In recent times, standards compliance of browsers has reached the point where, as long as you write valid code, all the major browsers should interpret your code the same. This is almost always the case for HTML anyway, with other standards having a few more differences in support here and there.

But what if you pass a browser invalid code? What happens then? The answer is that the browser error handling comes into play to work out what to do with the code. It basically says "ok, this code doesn't validate, so how do we present this page to the end user? Let's fill in the gaps like this!"

It sounds great doesn't it? If you leave a few errors in your page, the browser will fill in the gaps for you? Not so, as each browser does things differently. For example:

```
<p><strong>This text should be bold</p>
<p>Should this text be bold? How does the HTML look when rendered?</p>

<p><a href="#"></strong>This text should be a link</p>
<p>Should this text be a link? How does the HTML look when rendered?</p>
```

The errors are that the `strong` element is incorrectly nested across multiple block elements, and the anchor element is not closed. When you try to render this across different browsers, they interpret the code in very different ways:

- Opera makes the subsequent elements children of the bold element.
- Firefox adds extra bold elements between the paragraphs, which were not present in the markup.
- Internet Explorer puts the text “This text should be a link” outside the anchor tag that creates the link.

This original version of this example can be found in Hallvord Steen’s article “[Same DOM errors, different browser interpretations](#)”—read this for a much deeper treatment of HTML errors, and some information about debugging tools.

None of the different browsers’ behaviours is incorrect; they’re all trying to fill in the gaps of your incorrect code. The bottom line is, *avoid invalid markup if at all possible in your page!*

Quirksmode

Another thing you should know about is *Quirksmode*. This is the mode the browser goes into if it encounters a page that has an incorrect doctype, or no doctype at all. In this mode, the browser takes a best guess as to what set of rules it should validate the code to, and again fills in the blanks as best it can. This mode exists really to allow older pages to still be displayed, and should never be used when authoring a new page.

How to validate your pages

Now we’ve explored all the theory behind validating your HTML, I’ll talk about the easy part—the actual validation! Ok, that’s not completely accurate. Putting a URL into a validator and seeing if the page is valid or not is easy; working out what is wrong and fixing the errors is sometimes not so easy, as the error messages can sometimes be a little bit cryptic. I’ll go through some examples below.

The example we’ll be looking at in this section is as follows (you can also [download or view the HTML](#)):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
  <head>
    <title>Validating your HTML</title>
  </head>
  <body>
    <h2>The tale of Herbet Gruel</h2>
    <p>Welcome to my story. I am a slight whisp of a man, slender and fragile,
features wrinkled and worn, eyes sunken into their sockets like rabbits cowering
in their burrows. The <em>years have not been kind to me</em>, but yet I hold no
regrets, as I have overcome all that sought to ail me, and have been allowed to
bide my time, making mischief as I travel to and fro, 'cross the unyielding
landscape of the <a href="http://outer-rim-rocks.co.uk" colspan="3">outer
rim</a>.</p>

    <h3>Buster</h3>
    <p>Buster is my guardian angel. Before that, he was my dog. Before that, who
knows? I lost my dog many moons ago while out hunting geese in the undergrowth. A
shot rang out from my rifle, and I called for Buster to collect the goose I had
felled. He ran off towards where the bird had landed, but never returned. I never
found his body, but I comfort myself with the thought that he did not die; rather
he transcended to a higher place, and now watches over me, to ensure my well-
being.

    <h3>My possessions</h3>
```

```
<p>A travelling man needs very little to accompany him on the road:</p>
<ul>
  <li>My hat full of memories</li>
  <li>My trusty walking cane</li>
  <li>A purse that did contain gold at one time</li>
  <li>A diary, from the year 1874</li>
  <li>An empty glasses case</li>
  <li>A newspaper, for when I need to look busy</li>
</ul>
</body>
```

This simple page consists of three headings, three paragraphs, one hyperlink, and one unordered list. It uses the XHTML 1.0 Strict doctype as its rule set to validate against. There are a few errors in the document, which you'll discover below using the W3C HTML validator.

The W3C HTML validator

As mentioned above, the [W3C has an online validator available](#)—navigate to this by right/ctrl-clicking on the hyperlink you see here and selecting the “Open in new tab” option—it’ll be useful to be able to switch tabs to get between the validator and this article as you go through this example.

Note that you can also validate pages in the W3C validator from directly within the Opera browser by simply right/Ctrl-clicking and selecting the “Validate” option.

You'll notice that the validator has three tabs available across the top of the interface:

- Validate by URI: Allows you to enter the address of a page already on the internet for validation
- Validate by File Upload: Allows you to upload an HTML file for validation
- Validate by Direct Input: Allows you to paste the contents of an HTML file into the window for validation

Whichever method you use should give you the same result; I think it's easiest to test the example page from here by copying the full example code from above, and pasting it into the third tab along. Doing so should give you the result shown in Figure 1:

The screenshot shows the W3C Markup Validation Service interface. At the top, it says "W3C® Markup Validation Service" and "Check the markup (HTML, XHTML, ...) of Web documents". Below this, there are tabs for "Jump To: Validation Output". The main content area has a red header that reads "Errors found while checking this document as XHTML 1.0 Strict!". Below this, a table shows the validation results:


Result:	17 Errors
Source:	<pre><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html xmlns="http://www.w3.org/1999/xhtml" lang="en"> <head> <title>Validating your HTML</title> </head> <body> <h2>The tale of Herbert Gruel</h2> <p>Welcome to my story. I am a slight whisp of a man, slender and fragile, features wrinkled and worn, eyes sunken into their sockets like rabbits covering in their burrows. The years have not been kind to me, but yet I hold no regrets, as I have overcome all that sought to ail me, and have been allowed to bide my time, making mischief as I travel to and fro, 'cross the unyielding landscape of the <a</pre>
Encoding:	utf-8 (detect automatically)
Doctype:	XHTML 1.0 Strict (detect automatically)
Root Element:	html
Root Namespace:	http://www.w3.org/1999/xhtml

Figure 1: The results of validating the sample document—17 errors!

This may sound worrying, especially when I tell you that there aren't 17 errors in the document! Don't despair—it is reporting more errors than there actually are because often an error at the top of the page will cascade, making the validator report more errors further down, as it looks like more elements are not closed or incorrectly nested. You just have to think about what the error messages mean, and look for obvious errors in the markup. Table 1 below shows all of the errors I fixed to make the page validate, along with my logic for working out what was wrong, and the fix I applied to solve the problem.

Table 1: The errors I fixed to make the example page validate		
Error message	Logic	Fix made
Line 8, Column 461: there is no attribute "colspan"	We know that there is a <code>colspan</code> attribute, and it is valid HTML, so why is it saying it doesn't exist? Wait, maybe it means it is being used on an element that you shouldn't use it on? Sure enough, it is being used on an <code>a</code> element—wrong!	Removed the <code>colspan</code> attribute from the <code>a</code> element.
Line 13, Column 7: document type does not allow element "h3" here; missing one of "object", "applet", "map", "iframe", "button", "ins", "del" start-tag . <h3>My possessions</h3>	Again, from first glance this seems strange—the <code>h3</code> element is properly closed, and allowed in this context. You should note that often, this error message means that there is an unclosed element nearby...	Added a closing <code>p</code> tag to the line above the heading in question.
Line 19, Column 40: document type does not allow element "li" here; missing one of "ul", "ol", "menu", "div" start-tag . A diary, from the year 1874	This one is pretty easy—you can see from the line it is pointing you to, at a glance, that the end <code>li</code> tag has a missing closing slash (<code>/</code>)	Added a closing slash to the line in question.
Line 23, Column 9: end tag for "html" omitted, but OMITTAG NO was specified . </body>	Again, it doesn't take much to work out that this means the end <code>html</code> tag is missing. The error message explanation even starts with You may have neglected to close an element.	Added the missing end <code>html</code> element.

With these errors fixed, the validator now gives a rather satisfying success message, as shown in Figure 2:

**Markup Validation Service**
Check the markup (HTML, XHTML, ...) of Web documents

Jump To: [Congratulations](#) · [Icons](#)

This document was successfully checked as XHTML 1.0 Strict!

Result:	Passed
Source:	<pre><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html xmlns="http://www.w3.org/1999/xhtml" lang="en"> <head> <title>Validating your HTML</title> </head> <body> <h2>The tale of Herbert Gruel</h2> <p>Welcome to my story. I am a slight whisp of a man, slender and fragile, features wrinkled and worn, eyes sunken into their sockets like rabbits covering in their burrows. The years have not been kind to me, but yet I hold no regrets, as I have overcome all that sought to ail me, and have been allowed to bide my time, making mischief as I travel to and fro, 'cross the unyielding landscape of the <a</pre>
Encoding:	utf-8 (detect automatically)
Doctype:	XHTML 1.0 Strict (detect automatically)
Root Element:	html
Root Namespace:	http://www.w3.org/1999/xhtml

Figure 2: A success message to say that all my errors have been fixed.

This is about all there is to it really. You just need to keep your wits about you, and remember what doctype your page is being validated against.

Summary

After reading this article, you should be comfortable with using the online W3C validator to validate your HTML. This really is the tip of the iceberg with regards to validation—there are more complicated tools listed below, which will help you out as your pages start to get larger and more complicated.

Further tools to check out

- [The Opera debug menu](#)
- [General validation bookmarklet](#)
- [The Firefox web developer toolbar extension](#)
- [The IE developer toolbar](#)
- [Safari tidy](#)
- [HTML tidy](#)

Exercise questions

- What happens when a browser parses invalid HTML?
- What is the problem with this?
- Will using a frameset in a document validated against the HTML 4 Strict doctype generate an error?

About the author



Photo credit: [Andy Budd](#).

Mark Norman Francis has been working with the internet since before the web was invented. He currently works at Yahoo! as a Front End Architect for the world's biggest website, defining best practices, coding standards and quality in web development internationally.

Previous to Yahoo! he worked at Formula One Management, Purple Interactive and City University in various roles including web development, backend CGI programming and systems architecture. He pretends to blog at <http://marknormanfrancis.com/>.

25: Accessibility basics

BY [TOM HUGHES-CROUCHER](#) · 26 SEP, 2008

Introduction

When you create a web site, accessibility—making the web site usable by everyone, regardless of their ability or disability—should always be a central concern. So far in this course accessibility has always been implicit in all the examples you’ve seen, even if you didn’t realise it; in this article I’ll now look at accessibility explicitly, so you can understand fully what it is, why it is important, how to ensure that sites are accessible, and what guidelines exist to define accessible sites. The table of contents is as follows:

- [What is accessibility?](#)
- [Why is accessibility important?](#)
 - [The legalities of accessibility](#)
 - [Potential markets](#)
 - [Search engines](#)
 - [Ethics and branding](#)
- [Designing with accessibility in mind](#)
- [Interoperability requirements](#)
- [Features of an accessible web page](#)
 - [Semantic structure](#)
 - [Alternative content](#)
 - [Defining interaction](#)
- [Standards for accessibility](#)
 - [Web Content Accessibility Guidelines 1.0](#)
 - [Web Content Accessibility Guidelines 2.0](#)
 - [Section 508](#)
 - [Other standards](#)
- [Summary](#)
- [Exercise questions](#)

Before I look specifically at accessibility on the web, let’s start by looking at accessibility in general terms—after all, accessibility isn’t just a concern associated with web sites; it is potentially a concern with every service, object or technology you’ll come across in life.

Note that an associated topic to learn about is [WAI ARIA](#)—the Web Accessibility Initiative’s Accessible Rich Internet Applications initiative, which is basically a methodology to allow the creation of more accessible Ajax/JavaScript-powered applications. You can find a [great introductory article covering ARIA on dev.opera.com](#).

What is Accessibility?

Look around. Hopefully you will see some other people; if you don’t, why not take a quick walk? You’d probably enjoy it and it would do you good. None of the people you will see around you are the same—some have brown hair, some don’t. Some have blue eyes, some don’t. Some wear glasses, some don’t. None of us are completely alike. Some differences like hair or eye colour are cosmetic—they don’t significantly affect the way we live our lives. Some differences, like wearing glasses, do. Accessibility is a simple thing, a philosophy, although in some countries it is also part of the law.

Accessibility is treating everyone, no matter what their ability, the same.

I realise that this statement is open to interpretation. Most discussions of *accessibility* first talk about *disability*. This implies that people with a disability deserve special treatment. This isn’t what accessibility is about—it’s actually a symptom of the way people have traditionally built buildings, web sites, and pretty much everything else in fact.

When you build things with the assumption that everyone is the same as you, then they will always be wrong for some other people. People assume accessibility is about helping people with disabilities because retrofitted accessibility is very obvious in our societies. For example, a lot of buildings that started life with only steps have suddenly sprouted cheap ugly ramps. However, accessibility has long been a feature of military design. Why? because it is often critical for survival—at high g-forces jet fighter pilots can't do the same things they can do on the ground. If aircraft designers didn't take the needs of pilots in both high and low gravity environments into account then there would be a lot more plane crashes.

So, what does this mean for web site developers? The short answer is that you need to try to be more aware of the needs of the entire audience that might look at your site. A longer answer might require you to learn a little about the differing levels of ability people can have, and how they use computers. By applying the techniques outlined in this curriculum and other related articles you can create sites that work with many forms of interaction. Your web sites will be usable by people whether they:

- Are blind or severely visually impaired, and listen to web sites using a screen reader, or feel them on a braille display.
- Are shortsighted, and blow them up to 200% font size.
- Have motor disabilities so can't use their hands to manipulate a mouse, and therefore use a pointing stick to manipulate the keyboard, or an eye pointer to manipulate the web site.
- Use trackballs, or other more unusual types of computer control system.

Don't worry about the specific details of these interactions— we'll go through those step by step next.

Why is Accessibility important?

Accessibility is important for one big reason and a whole lot of little ones. The main one is that *we are all different and yet we all have an equal right to use web sites*, but there are lots of other reasons why you should make accessibility considerations a part of how you build web sites:

- In some countries it's the law.
- You don't want to exclude any potential customers/visitors from using your site.
- Accessible sites tend to rank higher on search engines.
- You are demonstrating good ethics—something that customers will value.
- Once you build web sites with web standards it hardly requires any extra effort to make it accessible, which gives so many benefits; there is also a lot of crossover between sites being more accessible, and sites being more compatible with mobile phone browsers—another circumstance that makes web site harder to use, although for different reasons. In fact, some work has been done on analysing the relationship between web accessibility and mobile web development best practices—see [the WAI "Web Content Accessibility and Mobile Web" page](#) for more.
- Techniques that help people with disabilities benefit all users.

Now I will move on to look at some of these points in more detail.

The legalities of accessibility

Note: it's important to understand the basics of the legal stuff but unless you are a lawyer and really know what you are talking about, you should take extreme care giving an opinion about legal issues.

In the UK, under the [DDA](#), it is illegal to discriminate against disabled people when recruiting and employing people, and providing services or education. Discrimination is defined as not making "reasonable adjustments" to support everyone, regardless of (dis)ability. This applies of course to making services or education available via the medium of web sites.

In the USA and European Union there are also requirements for Governmental web sites. In the USA, federal government (and some state government) web sites are expected to abide by [Section 508](#). Section 508 is a document that tries to define what the minimum requirements are to achieve accessibility. Section 508 covers more than just web sites; it also deals with any other technology that might be used by a federal employee. In Europe the European Commission has recognised the W3C's Web Accessibility

Initiative (WAI) and recommended it for use with all member states. The [WAI](#) produces guidelines for web sites, web authoring tool manufacturers and web browsers (for example, the WCAG, which I'll look at later on.)

Potential Markets

When you only make web sites (or anything else) for one specific type of person you are excluding other types of people even if you don't realise it, and these people can easily add up to a significant (if not a majority) market share. In 2000 the UK supermarket chain Tesco started a project to make a separate version of their online grocery site specifically targetting people with visual impairments. It was noted by Julie Howell of the RNIB that "Work undertaken by Tesco.com to make their home grocery service more accessible to blind customers has resulted in revenue in excess of £13m per annum, revenue that simply wasn't available to the company when the web site was inaccessible to blind customers." So if Tesco hadn't considered people with visual impairments they would have been missing out on a market of customers that was worth at least £13 million.

The core lesson here is that people of all abilities need the same services; groceries, taxis, electricity; and enjoy the same things; films, music, bars. Assuming that someone's personal situation in life changes their ability or desire to participate in all aspects of society has been shown to be a mistake time and time again.

Search Engines

Search engines are not people. Often when people build web sites they do it without considering how they are going to be found on Google, Yahoo, etc. Search engines are just computer programs, and they can only use information they can understand to index your site. This makes them much like the screen readers that a person with a visual impairment might use.

The most obvious example of how this affects web design is images. Computers display images by having a list of what colour every pixel is and sending that information to the monitor. If you put an image on a web page that contains some text, for example a logo, the computer has no idea what that text says or even that the image contains text. In HTML the image element contains a way to describe in text the contents of an image, the `alt` attribute. You should provide text to describe all non-decorative images on your site, and you certainly shouldn't represent whole paragraphs of text as images (or Flash for that matter)—blind people and search engines won't have a clue what the text says! As a result, your search engine ranking (ie how easy it is to find your web site using search engines such as Google) will suffer and you'll be needlessly missing out on a valuable market.

Ethics and branding

While everyone *should* support Accessibility, it doesn't mean that everyone does. By supporting accessibility you are acting in the best interests of the community. This is something you can be proud of—showing that you care about everyone in society can only enhance a brand image. As professionals it's our job to try and produce the best quality output we can. In a society that values us as individuals it's important to not exclude someone because they have different needs.

By making responsible choices in policy and genuinely demonstrating that you are implementing those policies you can create an extremely positive brand image. Companies that show that they care about their customers will retain much more loyalty than those that don't.

Designing with Accessibility in mind

The key to accessibility is thinking about a problem and knowing you are going to solve it for more than one kind of user. If you try to treat accessibility like something you can bolt on at the end then you will get a nasty-bolted-on-at-the-end thing. It'll take longer, won't work as well and look damned ugly.

The best way to achieve a well-engineered solution is to design with all the requirements in mind from the start. This doesn't mean you shouldn't change your plan or add some things you missed, but you should try

to be aware of what the complete problem you are trying to design for is. In the case of web sites this means creating a solution usable by all your users including those who may not be able to use a mouse, or a keyboard, or a monitor, etc.

Interoperability requirements

Interoperability requirements are especially likely to vary from situation to situation.

New technology is often introduced without support for accessibility. For example, Microsoft's new Silverlight plugin does not expose information via the accessibility APIs used by screen readers and other assistive technology, although such support is planned for the future.

Where support is theoretically present it can take time for assistive technology to make use of it. Newer screen readers work much better with JavaScript-triggered updates to HTML structures than older screen readers, for instance.

Even when long-established, accessibility support may continue to differ across platforms. For example, the Adobe Flash Player plugin has long exposed information to the Windows accessibility API, but not to the Apple or GNOME equivalents.

There also tends to be a lag between the arrival of supporting technology and its wide distribution. Whereas browsers and plugins nowadays tend to be free, whereas mainstream assistive technology can be very expensive. For example, one of the most popular screen readers is Freedom Scientific's JAWS for Windows. A new version comes out roughly every year. JAWS Professional retails for \$1,095, and even if you spend \$200 to get a Software Maintenance Agreement for the next two versions, upgrades will still cost \$500 or more. Consequently, although the latest release is version 9, you can still find lots of JAWS users using older versions.

So when you aim to build websites for the public Web, you need to take some account of interoperability with a highly varying client-side user/technology combination. There are four approaches:

- Progressively enhance your web site, testing for support as you go.
- Allow users to turn off problematic enhancements.
- Provide alternate versions with the same content or functionality.
- Advise your clients on what technologies they need to support and give examples of companies that do support those technologies.

Within intranets, backwards compatibility and variety is less of a concern. A given organization might be able to guarantee that all employees with disabilities will have access to assistive technology with decent support for DHTML, for example. In such circumstances, and with proper testing with the provided assistive technology, it would be reasonable to treat JavaScript as a baseline.

Forward compatibility and cross-platform compatibility are still issues however, so open, standard technologies should be preferred over proprietary, non-standard technologies.

For example, you might be developing an intranet training application for a large corporation. They have asked you to ensure the application is accessible, but have not specified a standard to which you must conform. You speak to their IT department and find out that everyone has the latest version of Internet Explorer, with JavaScript enabled, Flash installed and enabled, and will be provided with modern assistive technology they need to support those items. Now, even if the company moves to Unix-based platform, there will be assistive technology that supports JavaScript, but Flash text and controls are only accessible on Windows. You can safely make scripting and Flash a baseline requirement for your application. But you decide to only use Flash for playing video, and to build the control sets for the Flash video from web standards, since Flash controls are only accessible to assistive technology on the Windows platform. That way, the application would still be accessible even if the company migrated to Unix.

Organizational IT policies may change, and the best attempts to make JavaScript functionality operable and exploit the accessibility feature-sets of plugins may fail, so even if you have a technology baseline progressive enhancement from a core HTML layer is still a good idea.

Features of an accessible web page

In this section I will go through the different accessible features of a web site—that is, what an accessible web site should contain. I'll explain each one in detail.

Semantic structure

One of the foundations of web standards is the use of semantic structure in HTML. Semantic structure is also extremely important for accessibility. This is because it provides the framework for the information on the page. When people can't see the visual style of the page, the semantic structure helps to indicate a number of things to them. It can indicate their position in the hierarchy of the document and the ways they can interact with the different elements of the page, as well as providing emphasis to textual content in the right places.

A good example of how the semantic structure of a document is important to accessibility is navigation. A well structured navigation menu is a list of items. You can mark this up as an HTML list:

```
<ul>
  <li>Menu Item 1</li>
  <li>Menu Item 2</li>
  <li>Menu Item 3</li>
</ul>
```

By having navigation menus structured as lists it is easy to let someone using a screen reader, who can't see the list, know it's a list. This is because their screen reader tells them it's a list. If you don't use list markup then the screen reader has no way of knowing it's a list and telling the user.

You can find more information on how to use the correct semantics in your HTML in many of the earlier articles in the course, mainly the ones that cover HTML.

Alternative content

As mentioned in the [section about search engines](#), ensuring that there is an accessible alternative to content and navigation is essential. Text is considered the universal currency of content with [one caveat](#), as you'll see below. Text can be easily read aloud by a screen reader, made bigger or smaller, have its contrast easily altered and many other transformations. It's because it is so easy to manipulate text that more exotic forms of content should have a text-based alternative to them. Some formats, like the newer versions of Flash, have text access built into them so that the textual content within them can be accessed directly without needing to provide an alternative for the whole medium.

The one disability group that a text alternative can't necessarily support is people with cognitive disabilities. The difficulty with supporting people with cognitive disabilities is that often they require different content, rather than the same content in a different medium. This is not to suggest you shouldn't try. Simplifying the language and terminology used on your site benefits everyone. Groups like the [Plain Language Commission](#) have been advocating a "plain speech" approach to the material companies use to inform their customers of important information such as legal requirements and terms and conditions. They provide a [plain english lexicon](#) containing terms that can be used to help communicate effectively using the simplest language possible.

How should you implement text alternatives on your site? The first step is to identify things that aren't already text. In HTML there are only so many things that aren't already text. Images are the most obvious ones. Here is an example of an accessible use of an image:

```
<p>An interesting piece of art is Michelangelo's "God creates Adam"
```



```
.</p>
```

The image in this example is an integral part of the content. The `alt` attribute contains a short description of the image for people (or search engines) that might not be able to see the image correctly. The `longdesc` attribute allows you to link to an HTML page containing a full description of the image. This is generally only used to describe complex images that are used as central content. It also suffers from poor support in browsers. Most of the time you will only use the `alt` attribute.

When images are used for things other than content, such as navigation, or purely visual decoration you should treat them differently from content images. Images used to make buttons or page navigation look more attractive should have an `alt` attribute that matches the text in the image. The `alt` attribute simply functions as an easy way to allow the computer to read the text contained in the image (and hence read it to the user of a screen reader).

In the case of purely decorative images, images used for tracking adverts, or any other image that a user wouldn't be expected to be interested in or interact with, you should set the `alt` attribute to be empty. This does not mean omitting the attribute but setting `alt=""`. This is because of a tactic screen readers have used to help their users cope with very inaccessible pages. When an image doesn't have an `alt` attribute, especially when it's part of a link, the screen reader reads the URL of the image to the user. This is so they can guess what the image is from the URL, for example if the image is named something like `add_to_cart.gif`. Therefore, you should set `alt=""` on images that you know the user won't be interested in, so that screenreaders won't read out *every* image's URL, which could be rather frustrating for the screen reader user.

Not all forms of content are as simple as an image. More complex media like Flash (Flash files can be whole web sites in themselves) or movies require more complex descriptions. The most recent versions of Flash allow you to provide text alternatives for the items within the Flash movie, just like in HTML.

Defining interaction

A lot of today's web involves the use of technologies in addition to HTML. Even something as basic as CSS can be used in ways that make a page or interaction much less accessible. The key to accessibility in interaction is starting with the simplest interactions and using those as the building blocks of more complex interactions.

Note that the point of this example is to make you think about the role different things on web pages have. To ensure that they are accessible, they must be semantically sound in terms of both the HTML elements used, and the visual metaphor used. If you find this confusing, then reread the example a few times, and also look at a few menus and other web page components, while thinking about not only if the correct HTML is used, but also if the look and feel of the component successfully makes sense in terms of what its functionality is. You wouldn't expect a web page visitor to search using a text box labeled "enter your mail address to sign up for this newsletter", nor would you expect a sighted visitor to be able to find content of interest if all the headings were styled just like regular text (similarly, you wouldn't expect a blind user to find content of interest if all the "headings" were actually just paragraphs made to look bigger using CSS or font elements).

A good example of this is the commonly used visual metaphor of tabs. The tab metaphor is drawn from ring binders indexed by topic. This has been translated to computers to allow a single area on the screen to display the information from various topics represented by tabs connected to that area—you can see a good example of tabs on dev.opera.com—look at them at the top of the page. So far it is all reasonably simple. The problem lies in the technologies used to create tabs—they are often implemented using JavaScript.

As soon as tabs are used as part of an interaction more complex than allowing the user to select the information the original metaphor has been broken, but often the same code is still used to represent the tabs. In the example below the HTML shows what a tab control that displays information looks like:

```
<div class="tabcontrol">
  <div class="hd">
    <ul>
      <li><a href="#dogs" class="selected">Dogs</a></li>
      <li><a href="#cats">Cats</a></li>
      <li><a href="#fish">Fish</a></li>
    </ul>
  </div>
  <div class="bd">
    <p id="dogs" class="selected">Some information about dogs. The dogs tab is
the default tab.</p>
    <p id="cats">Some information about cats.</p>
    <p id="fish">Some information about fish.</p>
  </div>
</div>
```

In this example the `selected` class would be used to specify which tab needs to show the “tab to the front” graphic, for example check out the “Articles” tab at the top of this page which uses this method.

This structure is fine for informational content. In this example, the `class` of `selected` would be used to signify which tab is the active tab, ie, the one that is open and displaying its information; the others would be closed (ie, their paragraphs hidden) until their corresponding links are clicked on. The dogs tab is the default active tab, as shown in Figure 1.

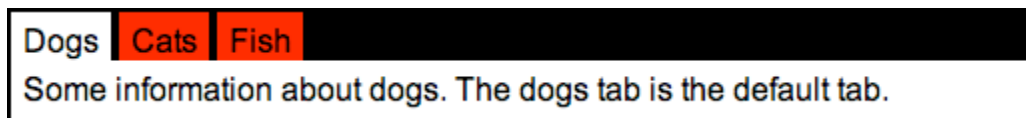


Figure 1: A simple tab control showing the dogs tab, the default, active.

Once another link is clicked on (as shown in Figure 2) JavaScript would then be used to dynamically move the `class="selected"` to that link, at which point style would be applied to that tab to display it, and the one that was previously displayed would be hidden.



Figure 2: Now a different link has been clicked on, its corresponding tab becomes active.

you will find real working examples of this kind of control in some of the later JavaScript chapters, to be published soon.

It has also become common to use tabs to allow users to select different kinds of searches. In this case the concept starts to break down if you try to reuse the style of code from the previous example:

```
<div class="tabcontrol">
  <div class="hd">
    <ul>
      <li><a href="#dogs" class="selected">Dogs</a></li>
      <li><a href="#cats">Cats</a></li>
      <li><a href="#fish">Fish</a></li>
    </ul>
  </div>
```

```
<div class="bd">
  <form id="dogs" class="selected" action="search.html"
method="GET"><div><label for="dogsearch"><input type="text" name="dogsearch"
id="dogsearch"><input type="submit" value="Search for Dogs"></div></form>
  <form id="cats" action="search.html" method="GET"><div><label
for="catsearch"><input type="text" name="catsearch" id="catsearch"><input
type="submit" value="Search for cats"></div></form>
  <form id="fish" action="search.html" method="GET"><div><label
for="fishsearch"><input type="text" name="fishsearch" id="fishsearch"><input
type="submit" value="Search for fish"></div></form>
</div>
</div>
```

Employing the same code structure no longer makes sense—in this case you get the same form elements repeated again and again in order to fit into the concept of replacing content, which is a waste of markup. Instead of thinking visually it's important to think about the interaction itself. In this example, rather than selecting new information to view the tabs you should be changing the interaction the user has with the search form. In effect the only thing a tab should be doing is selecting what type of animal the user is searching for. If you put this into practice you can create a much better interaction for all users of the site, with cleaner, easier to maintain markup:

```
<form action="search.html" method="GET">
  <fieldset>
    <legend>Search within:</legend>
    <ul>
      <li><label for="dogs">Dogs</label><input id="dogs" type="radio"
name="animal" value="dog" checked></li>
      <li><label for="cats">Cats</label><input id="cats" type="radio"
name="animal" value="cat"></li>
      <li><label for="fish">Fish</label><input id="fish" type="radio"
name="animal" value="fish"></li>
    </ul>
  </fieldset>
  <input type="text" id="searchfield" name="search">
  <input type="submit" value="Search">
</form>
```

By creating the interaction first, the markup is cleaner and all users of the site get the best experience possible. When we started by extending a visual metaphor we quickly broke the interaction and created some horrible markup based on the assumptions of the previous example. Had we been using AJAX to insert the content instead of having it all on the page, it would have been even worse. Then users without JavaScript would have had to load an entirely new page to get a search form for cats or fish. By thinking about the basic interaction first (rather than the visuals) you can simplify the problem. Now we can still maintain a tab metaphor (albeit with a bit of styling and script) while only using one form for all the searches.

This is the core to understanding how to do accessible interaction. One of the great things about HTML is that the hard work of figuring out how to make the interactions in HTML accessible has already been done. As long as you don't use the technologies on top of HTML to break the metaphor you can make most things work for most people without much effort.

Standards for Accessibility

In this section I will review some of the standards and guidelines available that aim to define web accessibility, and help web developers to create accessible sites. Most of these systems include some kind of check list system so developers can check how their sites match up to different accessibility criteria.

Web Content Accessibility Guidelines 1.0

The W3C is one of the primary standards bodies on the internet. Their Web Accessibility Initiative (WAI) published the first version of their guidelines for making web sites accessible in May 1999. The Web Content Accessibility Guidelines (WCAG) are the most widely used standard for Accessibility on the Web. The use of WCAG 1.0 has been suggested or mandated by a number of governmental bodies including the EU and the Italian government.

WCAG 1.0 is a set of 14 Guidelines that try to encapsulate the goals necessary to achieving an accessible page. Within each guideline are a number of checkpoints, which are the real meat of the document. While the guidelines explain the concepts the authors had in mind, the checkpoints are what are used to validate conformance. Each of the checkpoints are ranked from priority 1 to priority 3, to illustrate how important they are. In order to conform to WCAG 1.0 you must complete every priority 1 checkpoint. Adhering to all of the priority 1 checkpoints gives you a conformance rating of “A”. If you meet all the priority 2 checkpoints *as well* then you conform to “AA”. Should you meet all the priority 1, 2 and 3 checkpoints then you would conform to “AAA”, which is the highest rating.

In reality WCAG 1.0 is a little old-fashioned. A lot of companies start by conforming to level “A” or “AA” and then look to add other guidelines such as the RNIB’s See it Right. WCAG 1.0 is a good starting point, but you should be looking towards some of the newer standards, especially if you use a lot of JavaScript, or other technologies that matured after 1999, when the WCAG 1.0 was released.

Another important thing to note about the WCAG 1.0 standard is it was designed to be part of a suite of 3 documents. Another one covered “User Agents”, which describes browsers (like Opera) and any extra technology people might need to use the web (like screen readers). The third covered authoring tools such as Dreamweaver or content management systems—it aims to try to make these tools do more of the work of making pages accessible. Unfortunately this vision hasn’t played out and the only standard out of the 3 to be widely adopted was WCAG 1.0. This means that often the expectations of WCAG 1.0 regarding user agents aren’t met, and very little of the burden of making web sites accessible is taken off you by authoring tools. This doesn’t mean you shouldn’t use WCAG 1.0; it simply means that it only meets part of accessibility and is not a complete solution.

Web Content Accessibility Guidelines 2.0

Since publishing WCAG 1.0, the W3C has been working on WCAG 2.0. This updated version of the standard is still in draft at the time of writing. Subject to the process of the W3C it will probably be a published standard by early 2009.

WCAG 2.0 is slightly different in it attempts to be more technology-agnostic than WCAG 1.0, ie it could be applied to HTML, CSS, Flash, etc. WCAG 2.0 is based on 4 principles for accessibility. These are:

Perceivable

People can access the content through a medium available to them. For example people who can’t see should be able to hear the content

Operable

People can interact with the web application or content.

Understandable

The content and user interface is understood by the people who are using it.

Robust

Any solution provided should be using widely available on different platforms or systems. This is to stop people inventing solutions that the majority of people wouldn’t be able to use because the hardware/software is restricted or prohibitively expensive.

It's important to note that web sites aren't expected to fulfill all of these requirements. The technology a user has should do some of the work too. For example it is expected that a screen reader will read pages to people who need it, rather than every web site providing an audio version of the content. However, the web site is expected to provide pages that can be read using common screen reading technology in order to make this possible. This difference is important, as it's the difference between a web site with an "accessibility widgets" (like a button to make the fonts a bit bigger) and a web page that will work in a multitude of different situations (eg varying browsers and devices that would be impossible to anticipate).

WCAG 2.0 also differs from WCAG 1.0 in the approach to technology. Since the standard is more technology-agnostic and deals with concepts about accessibility rather than concrete technical details it's important to pay attention to the surrounding documents to the standard. The ["standard" document](#) of WCAG 2.0 will provide the understanding but the ["techniques" document](#) provides solid implementable pieces of information for the developer. This is broken up into "general" techniques (technology ambiguous) and specifics for individual W3C technologies. The W3C doesn't write documents for proprietary technologies so you'll have to find techniques for technologies like Flash and Silverlight from other sources.

Section 508

[Section 508](#) is an extension to the American Workforce Rehabilitation Act of 1973. The version of Section 508 that became law in 1998 created a process that must be followed during US Federal government procurement. This means that all government agencies in the USA that are funded by federal money must comply with the process and guidelines set down in Section 508. These guidelines cover both web accessibility and other accessibility issues relating to computers and electronic communication. Whatever else you might have heard, there is no federal law mandating the use of Section 508 beyond the organisations described above. However, some US states and companies use Section 508 to define "accessibility" for their own procurement processes.

The part of Section 508 that covers web accessibility is [Subpart B § 1194.22](#). Clause 1194.22 is split into 16 requirements labeled *a-p*. The first 11 requirements (*a* through *k*) are stated as being directly equivalent to parts of WCAG 1.0. The requirements and their equivalents in WCAG 1.0 are listed in a reference table in the section 508 document. All the other requirements of Section 508 should be met by WCAG 2.0 with one exception. Requirement *m* refers to Section 508 [Subpart B § 1194.21](#). This requirement has a partial equivalent in the Robust principle of `<cite>WCAG 2.0</cite>`.

As of the time of writing of this article there was an ongoing investigation into a [new version of section 508](#) by the Telecommunications and Electronic and Information Technology Advisory Committee (TEITAC). TEITAC presented its findings to the assessment board for Section 508 in April 2008.

Other standards

Another important standard being developed by the W3C is the [WAI-ARIA](#) standard. This stands for Web Accessibility Initiative—Accessible Rich Internet Applications. It is a suite of documents that defines how to make complex web applications that use technologies like HTML, JavaScript and AJAX accessible. This standard is officially supported by the upcoming/current versions of most of the major browsers in the market place: Opera 9.5, Internet Explorer 8 and Firefox 3.

There are many other standards for web accessibility too numerous to go into much detail about. The W3C maintain an excellent [list of international policies relating to web accessibility](#)—this is a great resource to help find the policy documents relating to your local government.

Summary

Accessibility is an important topic for both economic and social reasons. It is not a feature of a web site, but a measure of the quality it was built with. If you consider your site's audience as you are building it (and before) you will build more accessible pages with all the benefits that this brings. There are a number

of well-known guidelines to help you—by conforming to those guidelines you can ensure that what you have built meets expert criteria in making your pages accessible.

Exercise questions

- Give 3 reasons why it's important to build accessible web sites.
- Use the internet to research the accessibility laws in your country and make a list of any laws that you think would apply to your web sites. Make sure you include if they ask you to use any web standards such as WCAG or Section 508.
- Explain how accessibility is important to search engine optimisation.
- Create an example of an accessible use of alternative content using some of your own content, such as a photo.
- Use the internet to research how you would make a technology like Flash or Silverlight accessible and write a comparison between making them accessible, and how you make HTML accessible.
- Explain how you would design an interaction on a web page to be accessible. Create the step by step instructions for creating a tree control (you don't actually have to make it).

About the author



Tom Hughes-Croucher has been working in the Internet industry since he has been working. He has contributed to a number of standards on Web technology for standards bodies such as the World Wide Web Consortium (W3C) and the British Standards Institute (BSI). More recently he worked in the Digital Music business providing digital music solutions to well known UK brands like Tesco, Three telecom and Channel 4.

Tom now works for Yahoo! as a technical evangelist. Specialising in front-end web technology and RESTful web services he likes to promote best practice wherever he can. Before that he looked after the European Frontpages serving many millions of European visitors a month—he just doesn't find scaling scary any more.

26: Accessibility testing

BY [BENJAMINHAWKESLEWIS](#) · 26 SEP, 2008

Introduction

Web accessibility testing is a subset of usability testing where the users under consideration have disabilities that affect how they use the web. The end goal, in both usability and accessibility, is to discover how easily people can use a web site and feed that information back into improving future designs and implementations.

Accessibility evaluation is more formalized than usability testing generally. Laws and public opinion frown upon discriminating against people with disabilities. In order to be fair to all, governments and other organizations try to adhere to various web accessibility standards, such as the US federal government's [Section 508 legislation](#) and the W3C's [Web Content Accessibility Guidelines \(WCAG\)](#).

However, it is important to distinguish between complying with a standard and maximizing the accessibility of a web site. Ideally, the two would be the same, but any given standard may fail to:

- address the needs of people with all disabilities.
- balance the needs of people with differing disabilities.
- match those needs to optimal techniques.
- use clear language to express needs or techniques.

Such weaknesses can lead those with good intentions astray and may be exploited by those seeking to rubberstamp inaccessible products.

Moreover, web accessibility is a goal, not a yes/no setting. It is a nexus of human needs and technology. As our understanding of human needs evolves and as technology adapts to those needs, accessibility requirements will change as well and current standards will be outdated. Different websites, and different webs, serve different needs with different technology. Voice chat like Skype is great for the blind, whereas [video chat is a boon for sign language users](#).

Disabilities pose special challenges when working out how easy a product is to use, because they can introduce additional experience gaps between users and evaluators. Accessibility evaluation must take account of what it is like to experience the web with different senses and cognitive abilities and of the various unusual configuration options and specialist software that enable web access to people with particular disabilities.

If you are trying to evaluate the usability or accessibility of your web site, putting yourself in the place of a film-loving teenager or a 50-year old bank manager using your site is difficult, even before disabilities are considered. But what if the film-loving teenager is deaf and needs captions for the films she watches? What if the 50-year old bank manager is blind and uses special technology (like a screen reader) which is unfamiliar to the evaluator in order to interact with his desktop environment and web browser?

Accessibility guidelines and tools help bridge these experience gaps. However, they are a supplement, not a replacement, for empathic imagination, technical ingenuity, and talking to users.

In this article, I will discuss approaches to evaluating web accessibility, both from the perspective of establishing formal compliance and from the perspective of maximizing accessibility. The article's structure is as follows:

- [When should testing be done?](#)
- [Understanding your requirements](#)
 - [External requirements](#)
 - [The details of conformance](#)
 - [Exceeding expectations](#)
 - [The importance of the user interface](#)
 - [Personas with disabilities](#)

-
- [Choosing an accessibility standard](#)
 - [The spirit of the law](#)
 - [Who should test?](#)
 - [Expert testing](#)
 - [Semi-automated accessibility checkers](#)
 - [Structural inspectors](#)
 - [Screening and using end-user assistive technology](#)
 - [Detailed inspection](#)
 - [Perceivability](#)
 - [Operability](#)
 - [Understandability](#)
 - [Robustness](#)
 - [User testing](#)
 - [Recruiting testers](#)
 - [Practical considerations](#)
 - [Choosing tasks](#)
 - [Interpreting the results](#)
 - [Communicating the results of accessibility testing](#)
 - [Summary](#)
 - [Exercise questions](#)

When should testing be done?

“Test early, test often” is an old software engineering saying. Tacking on testing at the end of the development process has two risks:

1. Projects tend to run over-time and over-budget. Testing is often rushed, omitted, or ignored thanks to such pressures.
2. It is more work to fix problems discovered late in a process than to do things right from the start. So to ensure quality and save time and money, accessibility evaluations should start right at the beginning of product design and be included in subsequent development iterations through to final delivery.

Understanding your requirements

Before you begin to evaluate a project for accessibility, you need to determine what the key requirements are for that project, given its environment, intended audience, and resources. Some requirements will be set by third parties like governments and clients; some you may be able to choose for yourself.

External requirements

Often requirements come from external sources, such as:

- Governments. This typically takes the form of general legislation against discriminating against people with disabilities, rather than mandating a particular standard or enumerating precise conformance requirements. An important exception is when legislation enforces a particular standard for public sector. For example, [Section 508](#) is a piece of US federal legislation, which mandates that websites produced for federal agencies must conform to at least a specific set of defined requirements. WAI's [Policies Relating to Web Accessibility](#) page provides a partial list of similar legislation. But to get an authoritative statement of the obligations in your jurisdiction, consult a lawyer.
- Customer policies. For example, [Shell currently try to ensure their websites conform to the "Double-A" conformance level of WCAG 1.0](#), so if you were developing a website for Shell you would need to meet (at least) the same standard.
- Marketing utility. Compliance with a particular standard, such as Section 508, might help sell a project to clients concerned about accessibility.

-
- Internal accessibility policies at your organization. For example, projects produced by the BBC need need to comply with the [BBC's Accessibility Guidelines v1.3](#).

The details of conformance

It is important to get as much clarity about external requirements as possible. Some accessibility standards have more than one possible level or type of conformance, so it is particularly important to nail down which is required. For example, WCAG 1.0 has three conformance levels:

1. People with some disabilities “will find it impossible to access information” in a document that does not pass level “A”.
2. People with some disabilities “will find it difficult to access information” in a document that does not pass level “Double-A”.
3. People with some disabilities “will find it somewhat difficult to access information” in a document that does not pass level “Triple-A”.

Draft WCAG 2.0 has three levels too, but the conformance possibilities are more complicated. Where a resource is part of a series of resources presenting a process (eg product discovery, selection, checkout, and purchase confirmation for an online store), the conformance level for all resources in the series is that of the resource with the lowest level.

Conformance claims must be based on “accessibility-supported” content technology. To be an accessibility-supported content technology, a technology must:

- Have been demonstrated to work with users’ assistive technology.
- Have user agents (browsers, plugins, etc.) that work with the users’ assistive technology and are available to users with disabilities at no cost above that for a user without a disability.

Note that within an intranet setting, you might be able to guarantee that such user agents would be available to users whereas you cannot guarantee the same thing on the World Wide Web. For example, an application might be usable without any commercial technology, but only be accessible to screen readers with a commercial plugin for which the organization has a site licence. That application could conform to WCAG 2.0 when deployed on the organization’s intranet, but not when deployed on the public Web.

WCAG 2.0 also allows more limited statements of conformance. An inaccessible resource can conform if an accessible alternative is provided. Publishers can make a statement of partial conformance where content is aggregated from other sources.

Exceeding expectations

Determining external requirements should only be the beginning of the process; they should be treated as a minimum set of requirements to which further goals should be added to maximize accessibility. As the person evaluating accessibility, it is your role to raise additional accessibility concerns, as you are the subject expert.

You may need to distinguish the two when delivering a final report. For example, a customer brief for an online supermarket might mention that they want a store accessible to blind users. Given the intended audience, you should also evaluate whether the store is accessible to users with other disabilities.

Note that external requirements for compliance with a particular standard do not necessarily prevent best practice guidelines from other standards being applied. For example, you might be evaluating a website for a web federal agency intended for use by senior citizens and be required to comply with Section 508. Section 508 stipulates that:

§ 1194.22 (c) Web pages shall be designed so that all information conveyed with color is also available without color, for example from context or markup.

This provision helps users who know how to customize the presentation of web content, but doesn’t maximize the accessibility of the default presentation of that content to the target audience by ensuring that there is sufficient contrast between suggested colors. Fortunately, there’s nothing stopping a web site

from fulfilling this requirement but also meeting the following Level provisions from the Web Content Accessibility Guidelines 2.0 draft:

1.4.3 Contrast (Minimum): Text and images of text have a contrast ratio of at least 5:1, except for the following: (Level AA)

- Large Print: Large-scale text and large-scale images of text have a contrast ratio of at least 3:1;
- Incidental: Text or images of text that are part of an inactive user interface component, that are pure decoration, that are incidental text in an image, or that are not visible to anyone, have no minimum contrast requirement.

Note: Success Criteria 1.4.3 and 1.4.6 can be met via a contrast control available on or from the page.

1.4.6 Contrast (Enhanced): Text and images of text have a contrast ratio of at least 7:1, except for the following: (Level AAA)

- Large Print: Large-scale text and large-scale images of text have a contrast ratio of at least 5:1;
- Incidental: Text or images of text that are part of an inactive user interface component, that are pure decoration, that are incidental text in an image, or that are not visible to anyone, have no minimum contrast requirement.

Note: Success Criteria 1.4.3 and 1.4.6 can be met via a contrast control available on or from the page.

By contrast control, the criteria means that you should provide a way of changing the colours to a high-contrast variation.

WCAG 2.0 is being designed to have a high degree of backwards compatibility with other standards, especially WCAG 1.0 and Section 508.

The importance of the user interface

Consider the special importance of making the user interface of a web site accessible. Even if content is not available in a suitable form, an accessible user interface may help users identify content of interest and seek external help in converting it to a form they can use. For example, a hard-of-hearing individual might be pointed to a video of a talk on a video-sharing site without captions. Because the URL uniquely identifies that video and because they can still use the player to see the video however they could submit it to a third party, such as the free [Project readOn](#) service, for captioning.

Personas with disabilities

An ideal approach is to build key disabilities for your project right into your other [user personas](#): fictional users that act as archetypes for how particular types of users would use a web site. Let us say you are evaluating prototypes for a video sharing site and your personas include:

- 23-year-old James Smith, who is football-mad and especially wants to share sporting highlights with friends.
- 34-year-old Sarah Maddison is a working mom who might not normally have time for a video sharing site. But her three-year old daughter is really keen on watching videos, and Sarah wants to sit and help her find suitable videos she wants to watch.

You can take these personas and incorporate disabilities including (for example):

- Impaired vision.
- Colorblindness.
- Blindness.
- Deafness.
- Hard-of-hearing.
- Deafblindness.
- Epilepsy.
- Dyslexia.

For example, you might decide that James is also deaf and wants commentary on match videos to be captioned, and Sarah has poor eyesight and struggles to read fancy fonts and tiny text. These personas guide your rejection of prototypes that fail to include the facility for closed captions in the video player, or use elaborate text headings that would require images.

The WAI's [How People with Disabilities Use the Web](#) and Shawn Lawton Henry's [Just Ask: Integrating Accessibility Throughout Design](#) contain some more example disability-inflected personas to get you started.

It shouldn't need saying, but don't assume people with disabilities are interchangeable. Disability is an incredibly varied phenomenon, and on top of that people with disabilities have all the variety that people without disabilities have, differing (for example) in gender, age, interests, values, and skills (perhaps most relevantly, in their computing expertise).

Again, comparing products against accessibility guidelines can help fill in the gaps that your personas do not cover. For example, perhaps you're following WCAG 2.0 with the video sharing site, but your personas don't include a user with epilepsy. Nonetheless, you read Guideline 2.3 ("Seizures: Do not design content in a way that is known to cause seizures") and decide that the system needs to be able to screen uploaded videos for flashing before displaying them.

Choosing an accessibility standard

If you need to choose an accessibility standard in order to manage web accessibility concerns across a team or in simply to guide you while testing, I'd advise looking at WCAG 2.0 because it:

- is designed around core human needs that are applicable to technologies other than HTML and CSS (such as Flash).
- carefully documents the reasoning for each conformance criterion.
- suggests practical techniques for meeting conformance criteria using current technologies.
- ensures each provision is testable.
- incorporates more recent research than current alternatives.
- is designed to be broadly compatible with existing accessibility standards.
- will be an international standard.

You can cite compliance with a specific draft of WCAG 2.0; for marketing purposes however it is best to also seek compliance with finished standards like Section 508 and WCAG 1.0 as well as that draft.

The spirit of the law

When testing against guidelines, it's important to keep in mind the underlying rationale for any specific technical guidance: to comply with the spirit, not just the letter, of the law.

Here's a cautionary tale. Section 508 (§ 1194.22) includes a requirement that says: "A text equivalent for every non-text element shall be provided (eg, via `alt`, `longdesc` or in element content)." Likewise WCAG 1.0 includes a checkpoint that reads:

Provide a text equivalent for every non-text element (eg, via `alt`, `longdesc`, or in element content). This includes: images, graphical representations of text (including symbols), image map regions, animations (eg, animated GIFs), applets and programmatic objects, ascii art, frames, scripts, images used as list bullets, spacers, graphical buttons, sounds (played with or without user interaction), stand-alone audio files, audio tracks of video, and video.

Unfortunately, many people reading such guidance misunderstand what a genuine text equivalent for a spacer and decorative elements should be, and produce markup like this:

```

```

In fact, since these images convey no new information and have no functionality, the right text equivalent for those images would be an empty string (`alt=""`), which causes the screenreader to just skip over the

alt attribute and not read it out. It is very annoying for a screenreader user to have to listen to text such as "fancy border" read out over and over again, when it does not provide them with any useful information.

WCAG 2.0 tries to be clearer. The [equivalent guideline](#) says: "All non-text content has a text alternative that presents equivalent information, except for the situations listed below". One of those situations is: "Decoration, Formatting, Invisible: If it is pure decoration, or used only for visual formatting, or if it is not presented to users, then it is implemented in a way that it can be ignored by assistive technology." Equally importantly, WCAG 2.0 tries to detail the reasoning behind the [guideline](#):

The purpose of this guideline is to ensure that all non-text content is also available in text. "Text" refers to electronic text, not an image of text. Electronic text has the unique advantage that it is presentation neutral. That is, it can be rendered visually, auditorily, tactilely, or by any combination. As a result, information rendered in electronic text can be presented in whatever form best meets the needs of the user. It can also be easily enlarged, spoken in a voice that is easy to understand, or rendered in whatever tactile form best meets the needs of a user.

Who should test?

There are basically two groups who conduct testing: experts and users.

Expert testing is important because experts understand how the underlying web technologies interact, can act as a clearing house for knowledge about different user groups, and have the inclination to learn dedicated testing tools.

User testing is crucial because users are the real experts in their own abilities and their own assistive technology. User testing can also reveal usability gaps between more and less technical users, and between people who are familiar with the web site in question (such as the expert testers themselves) and people who aren't (new users).

A web developer who knows how to use a screen reader is unlikely to explore a site the same as a regular screen reader user; screen reader users who program their own scripts are unlikely to explore the site using the same strategies as screen reader users who just do ordinary computing tasks like writing emails.

Knowledge gained in user testing is fed back into the expert testing process the next time testing is performed (either in another testing iteration on the same project, or a different project entirely). User testing also has a more subtle advantage. By humanizing accessibility and bringing developers together with end users, it can increase the motivation to build accessible websites.

Expert testing

There are four components to expert testing:

- Tool-guided evaluation: where a tool looks for accessibility problems and presents them to the evaluator (this would include accessibility checkers and code linters).
- Screening: where the expert simulates an end-user experience of the web site. Often you don't need to look very far to find accessibility problems. You might do no more than load the page in your browser and notice the text is very hard to read.
- Tool-based inspection: where the evaluator uses a tool to probe how the various bits of a web site are working together.
- Code review: where the evaluator looks directly at the code and assets of a web site to scour for problems.

While beginners may be especially dependent on tool-guided evaluation, evaluators of all levels of experience can benefit from each component. Even beginners can spot `img` elements without text equivalents in HTML markup, and as you get more experienced, you will get quicker at spotting problems before you progress to more rigorous testing. For experts on larger projects, it may not be feasible to manually review all client-side code or inspect all parts of a website, but a tool-guided evaluation can find

areas of particular trouble that deserve a closer look. Also, human evaluators may overlook things that a machine evaluation would have caught.

Unfortunately, although there are lots of accessibility tools, most of them are flawed in one way or another. For example, one tool that lists headings in HTML documents makes the error of not including `alt` text from `img` elements. Just as you should keep the spirit of the law in mind with standards compliance, so you should keep it in mind when using tools. Before complaining to someone about an accessibility problem, make sure it is a genuine issue not a tool error.

Semi-automated accessibility checkers

Once the first-glance problems have been fixed, a good next step is to throw the page at a semi-automated accessibility checker tool. If you are evaluating compliance with a particular standard, you will probably want to pick one that is designed for use with that standard.

If you're evaluating compliance with Section 508 or WCAG 1.0, [Cynthia Says](#) is a reasonable choice. If you're testing against German BITV 1.0 Level 2, the Italian Stanca Act, or the WCAG 2.0 draft, the only current option is the experimental [ATRC Web Accessibility Checker](#).

Such tools have significant limitations. There is no such thing as fully automated accessibility testing. For example, given the primitive nature of current artificial intelligence, a computer program cannot have the final say in whether some text is a genuine equivalent for a photograph in context. Even with areas that can theoretically be fully automated, checker programmers may err in their interpretation of accessibility guidelines and lose the spirit of the law amongst its letters.

Good tools inspect the page for accessibility problems and produce a list of things they judge to be errors and other things they judge worth human investigation. For example, if Cynthia Says finds an `img` element with `alt=""`, it will issue a warning (not an error!) instructing the user to “verify that this image is only used for spacing or design and has no meaning.” If the correct text equivalent for that image is an empty string, you should move on to the next error or warning.

Perhaps the biggest advantage of accessibility checkers is that if you choose one, such as [TAW 3](#), that can be run against multiple URLs, you can find pages in large collections that are likely to require closer inspection.

Structural inspectors

Many inspection tools are designed to probe structures of web content. Structures, in simple terms, define what the components of a web site are and how they relate to one another. For example, in the HTML document object model (DOM), text can be designated as a label for a form field using the `label` element. Browsers parse the HTML into a document object model. The browser associates various behaviour with particular components. For example, if you click the label of a checkbox, it will normally get checked.

Desktop environments and applications support interactivity with screen readers, speech recognition software, and other assistive technology by providing a similar structure that represents the content and functionality available in the visual presentation. On Windows, the main structural system is called Microsoft Active Accessibility (MSAA), or [UI Automation on Vista](#). For example, a dialog has a series of related children, such as its title, its fields, its buttons, and their labels.

Typical assistive technology mostly deals with the browsers' and plugins' representation of web content in terms of these structural systems rather than processing web document object models directly.

There are inspectors for both desktop-level structures and web-level object models. On the desktop-level side of things, OS X comes with [Accessibility Inspector](#) and [Accessibility Verifier](#). Microsoft provides inspector tools for [Microsoft Active Accessibility 2.0](#) and [Microsoft Active Accessibility 1.3](#). [Accerciser](#) is available for the GNOME assistive technology-SPI API.

Tools for poking at the (X)HTML document object model include DOM Inspectors as seen in [Opera Dragonfly](#) and [Firebug](#) and accessibility tool bundles like the [Web Accessibility Toolbar for Internet Explorer and Opera](#) and the [ICITA Firefox Accessibility Toolbar](#).

DOM inspectors show you the tree of elements and attributes and text constructed out of the (X)HTML serialization, whereas web accessibility inspectors abstract particular components or relationships and list them. For example, they might list all fields with their labels or all headings or all links.

Digging into the accessibility model should not normally be necessary for (X)HTML, though you might also want to investigate that layer if you think a browser is representing a correct (X)HTML structure incorrectly to assistive technology. Instead, you will normally be checking (X)HTML structures directly.

Not all content can be inspected with DOM or web accessibility inspectors. Inspecting what is exposed to the desktop-level accessibility structures is important for checking what plugin content (media players, Flash content, and Java applets) is being exposed to assistive technology that uses those accessibility models.

In general, you should check that all controls are exposed in the model with the appropriate role (eg text boxes are text boxes, buttons are buttons), and the necessary properties.

Screening and using end-user assistive technology

Screening involves emulating the experiences of people with disabilities while testing. This might take the form of using assistive technology to interact with a site or attempting to restrict one's abilities in some manner. For example:

- Using a mouthstick to press keys while testing keyboard accessibility.
- Viewing a page with the Vischeck simulator, which attempts to present the page, images included, as people with different forms of colorblindness see it.
- Turning off a monitor while using a screen reader in conjunction with a browser.

Screening can help build developer appreciation for the needs of people with disabilities and can reveal fundamental design flaws. Use of assistive technology can clear up certain misconceptions about how they do and don't support and interact with web standards. For example, popular screen readers do not use styles suggested for the `aural` or `braille` CSS media types, instead attempting to represent the `screen` type presented by the visual browsers with which they interact.

Using assistive technology is not a task to be taken lightly, since a good understanding of how to use such systems may require a degree of immersion and training. There's a serious risk of creating new misconceptions. Developers might struggle to do something with a screen reader and assume that reflects a failing in the screen reader, when it really reflects their inexpertise with the tool. They might try to use the tool the wrong way, for example trying to read a page in sequence where a real screen reader user would hop around it using headings and other elements looking for points of interest. Or alternatively, they might fail to read the screen properly. Reading through a page you can see or know well with a screen reader is very different from exploring a brand new site you cannot see.

Use of assistive technology needs to be accompanied by experience of how everyday users employ the technology and conclusions drawn from such use should ideally be confirmed with expert users. On the whole, beginners are better off leaving use of assistive technology to user testers.

Detailed inspection

Once all genuine problems identified by your chosen checker tool have been fixed, you can move on to manual testing, probing, and review of the project.

WCAG 2.0 splits its best practice criterion into four principles. Content and functionality must be:

- Perceivable (for example, images should have text equivalents).
- Operable (for example, it should be possible to interact with a web site without a mouse and navigate it with a screen reader).

- Understandable (for example, copy should not be more complicated than it needs to be and the web site should operate in a predictable manner).
- Robust (for example, web sites should work interoperably with different user agents and navigation should be consistent).

In this section, I shall present some examples of how expert testers can evaluate how far content matches up to these principles. Please note this section is not intended as a substitute for a review of WCAG and its techniques.

Perceivability

One subset of perceivability problems revolves around the provision of alternative media of various types. You can test for text equivalents by turning off images and multimedia in your browser and looking at the page. But you'll need to take special care with the `img` and `input` elements. Normally, you are advised to give all purely decorative images blank `alt` attributes (`alt=""`) so that the screenreader will just skip them. However, in the case of:

- Images that are the sole content of links
- Form buttons

When these elements are given `alt=""` attributes, screen readers will commonly treat the image or button as if the `alt=""` attribute is missing, and attempt to provide one (for example, by reading out the URL of the image).

Therefore in these particular circumstances you must ensure that images inside links or buttons have an `alt` attribute that describes the link destination or button action, even though this is slightly repetitive.

Testing for equivalents synchronized with multimedia, such as captions and audio descriptions, can be done by digging into the preferences for your media player to turn on accessibility settings.

Another group of perceivability problems concerns the styling of the page. There are three areas to investigate here:

- Is the suggested presentation of the page reasonably accessible? For example, is there sufficient color contrast? Is the text comfortably large? In addition to squinting at the page yourself, you can use a tool such as [Juicy Studio CSS Analyser](#) to check background and foreground color combinations against formulae that purport to measure legibility.
- Can publisher suggestions for presentation be safely mixed with common user preferences aimed at making content more legible, like increased font size, zoom, and different default colors? Try increasing the text size by about 2-5 steps; don't worry if the results are not pixel perfect but do worry if the layout is so broken it renders the content hard to read. Try changing your color preferences and see what happens. If publisher CSS sets colors, it should explicitly set background and foreground together to ensure that the combination of unusual preferences and publisher styles do not result in unreadable or invisible text. Popular browsers allow users to enforce their own color preferences and turn off CSS background images. When you try this yourself, it can reveal misconceived CSS image replacement techniques that hide text off-screen, since the image will not be loaded but the text will be still be invisible.
- If publisher suggestions for presentation are discarded, is all the information communicated by such suggestions preserved in the web content for use by the default stylings of the user agent or user styling? Try turning off CSS and inspecting the document object model to check that headings are marked as headings and tables are used for tabulated data not layout.

Operability

Health and safety is a crucial, though rarely considered, part of making a website operable. But flashing content risks triggering fits in photosensitive epileptics. You can take a screen capture of your website in use and feed it into the [Trace Center Photosensitive Epilepsy Analysis Tool \(PEAT\)](#) to test if it has flashing content likely to pose a danger to your users. Obviously, this is an especially big concern if you are creating

a video sharing website. At the product design stage, you might look at including an automated screening process for uploads.

Beyond that, a good way to test the operability of websites is simply to try to see if you can access all essential content and functionality with different devices:

- Try using your site with just the keyboard. Is current focus always clearly indicated? Can all functionality be accessed by keyboard?
- Try using your site with a touchscreen device.
- Try navigating your webpage with voice commands using Opera for Windows and its Voice add-on, or Windows Vista Speech Recognition and Internet Explorer. (Note: dictation-quality commercial speech recognition has recently been introduced to Mac OS X in the form of MacSpeech Dictate, but there is currently no equivalent on the free *nix platforms.)

Screen readers and other assistive technology can make use of the semantic structure of (X)HTML to correctly associate content and to enable navigation of content. For example, screen readers can allow users to jump to the next occurrence of headings or other element type, or they can list all occurrences of a certain type. Correct use of `label` and `legend` elements allows assistive technology to associate labels with the correct form fields; correct use of `th` elements and `header`, `scope`, and `axis` attributes allows it to associate table headings with table data cells. Semantic structure may be evaluated with a document object model (DOM) inspector like the one found in Opera Dragonfly. Accessibility inspection tools like the Firefox Accessibility Extension can make such tasks easier by, for example, listing the headings on the page, or listing the attributes of form fields (quickly showing which ones are missing associated labels). See Figure 1 for an example.

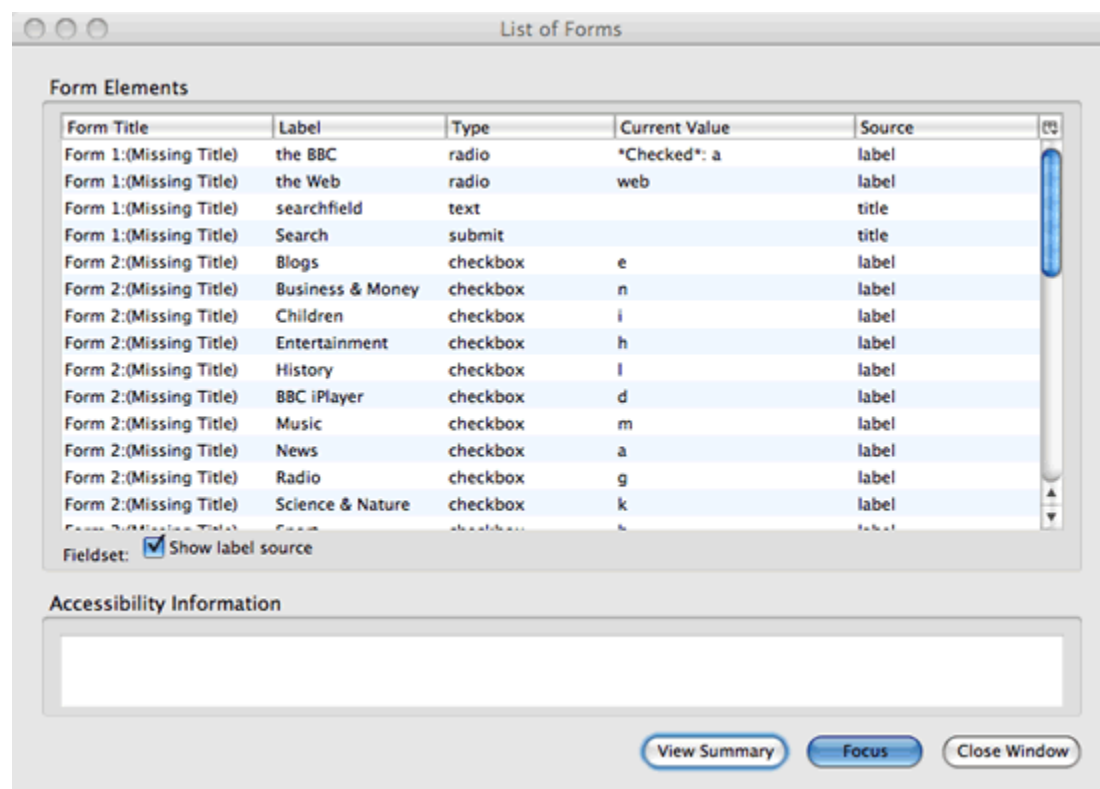


Figure 1: Screenshot of Firefox Accessibility Extension's forms information window with the new BBC homepage.

Understandability

Assessing comprehensibility is even more subjective than testing legibility. Unless an evaluator is new to a project or is a professional editor, they are probably not the best person to evaluate whether copy is as understandable as possible. You can however use [Juicy Studio's Readability Test tool](#) to get a rough idea of how simple your site's copy is.

Some aspects are highly objectively testable however, such as whether content has language metadata that allows (for example) screen readers and voice browsers to read content with the correct pronunciation. With HTML, you could use a DOM inspector to check for the presence of a `lang` attribute for the document and each change of language.

Keep an eye out for inconsistencies in web sites, both in terms of internal consistency and predictability from common web conventions. Screen magnifier users who only see part of a page at a time rely heavily on such consistency in order to know where to look to find given content and functionality.

Robustness

Testing whether content is robust involves checking that technologies are being correctly used. At a very basic level, you can run markup and code through linters such as:

- [WDG HTML Validator with warnings enabled](#)
- [W3C CSS Validator](#)
- [JSLint JavaScript linter](#)

Next, you can review code in depth to check that features are used correctly. For example, you can check that HTML native controls are used rather than faking controls with meaningless elements and JavaScript, and that JavaScript uses [feature detection rather than browser sniffing where possible](#).

Then you can test in multiple user agents and assistive technologies, checking the site is perceivable, operable, and understandable whatever combination of publisher CSS, JavaScript, and plugins are enabled or disabled.

The most common problem is probably obtrusive JavaScript, like anchors and buttons that are in the unscripted markup of the page but depend on JavaScript to actually do anything. But there are more subtle problems that arise from overly close coupling of JavaScript with other layers in the technology stack. For example, JavaScript might apply `CSS display: none;` to hide content, but what happens when publisher CSS is not applied?

Another example is multimedia controls. Often, when plugin content is included, the plugin's native user interface is disabled and the plugin is instead controlled by scripted HTML widgets. When the plugin content is only added via JavaScript after JavaScript-based plugin detection, this is fine. But sometimes plugin content is included in the pre-scripted state of the page. In such cases, it is worth checking not only that a fallback has been included in case a handling plugin is not available, but also that the native user interface of the plugin is not disabled unless JavaScript is available. If the former is not the case, then users will not see the fallback content at all; if the latter is not true then users will see the plugin but not be able to control it.

User testing

No amount of developer inspection and screening can substitute for the raw clash between a user and a web site. Given the difficulties of understanding all the subtle interactions between web content and assistive technology and the difficulties of approximating the experience of users with disabilities, this goes double for users with disabilities. If at all possible, you should test your site with real users with disabilities. This can be done on a large and expensive scale, but do not underestimate the benefits of doing even small-scale user testing.

Recruiting testers

Testers can be found in the same way as you find candidates for usability testing generally (eg through advertising and recruitment agencies). Your local disability organizations should be able to suggest appropriate forums for recruiting test subjects.

Testing is real work and should ideally be compensated as such. A rate of around 70 USD for an hour's testing is fairly common for user testing.

Having said that you may be able to find people who will test smaller projects for free. There will be people with disabilities among your friends, relatives, and colleagues. In addition, there are online discussion groups dedicated to software accessibility issues, such as:

- [WebAIM Accessibility Discussion List](#).
- [Web Accessibility Initiative Interest Group Mailing List](#): a forum for discussion of issues relating to Web accessibility.
- [British Computer Association of the Blind mailing list](#): for discussing Information Communication Technologies (ICT) for visually impaired people.
- [Magnifiers Yahoo! Group](#).
- jfw@freelists.org: A mailing list for users of the JAWS screen reader.
- [GW-Info](#): A mailing list for users of the GW Micro Window-Eyes screen reader.
- [Dolphin software users Yahoo! Group](#).
- [NVDA users mailing list](#).
- [Thunder users mailing list](#).
- discuss@macvisionaries.com: A list about use of OS X by the blind.
- macvoiceover@freelists.org: Apple VoiceOver users.
- [Blinux-list](#): A list about the use of Linux by people who are blind and visually impaired.
- [GNOME Orca users](#).
- [Ai Squared Forums](#): Including users of the popular ZoomText magnifier.
- [Deaf-Macs Yahoo! Group](#): For deaf, hard-of-hearing, and Usher or deafblind Mac Users.
- [deaf-uk-technology Yahoo! Group](#): Deaf-related technology discussion.

Such groups typically welcome questions from web developers about the accessibility of their sites or particular techniques.

Practical considerations

Remember that the test environment itself needs to be accessible. For example, if you are preparing written test materials, you need to be prepared to offer these in alternative forms. The logistics of replicating the user's browsing environment at your normal testing location are complicated, so it may be more realistic to test at the user's home. Failing that, even completely remote testing can be valuable.

One particular consideration that is probably even more important for users with disabilities than other users is what technology they are familiar with. Assistive technology can add many layers of complexities to their computing experience, creating a big divide between novice computer users and old-hands, and dividing users into communities who might be very adept with their own setup but highly disorientated by unfamiliar technology. (Think of how hard users without disabilities that affect their use of computers find it to switch between Mac and PC!)

If you take a longtime user of the Window-Eyes screen reader, sit him in front of an unfamiliar machine with the JAWS screen reader installed and ask him to test a website, it's going to be very difficult to distinguish his problems with JAWS from his problems with your website. Given the significant differences between versions and given how users often customize their setup, it may be difficult even if you provide Window-Eyes! For this reason, unless you are specifically testing how well your website's accessibility will hold up in unfamiliar settings (eg in libraries or friends' computers), it is best to allow users to test with their own setup or something as close as possible to it.

Likewise, unless you specifically want to test novice users or expert users, you should aim to select users who have around a year's familiarity with using their current setup to access the web. Both assistive technology and the conventions of the web itself are non-trivial to learn. With novice users you will not

know whether problems arise from your site or are intrinsic to the learning process, and experts may have tricks up their sleeves that others don't.

Choosing tasks

It's incredibly instructive even to observe users simply exploring a website. As with any other user testing:

- Try setting the users some specific tasks to accomplish.
- Ask them what they think and listen to what they say.
- Pay attention to what they do, because that may differ from what they say: stated preference is a poor guide to performance.

When you design a site, you need to focus on the transactions users want to make with your site rather than the particular controls they need to use. Likewise, when accessibility testing, the tasks you set should (at least initially) reflect the real goals of a visitor using the site, rather than being focused on their interactions with particular controls. These transactions will typically be similar for people with and without disabilities.

For example, if you are testing a video sharing site for accessibility, do not begin by asking them if they can use particular controls ("That's the volume slider. Can you adjust the volume?"). Instead, give them scenarios and ask them to achieve key user tasks. For example:

- Browse videos and choose one to play.
- Search for a video.
- Upload a video.
- Pause the video, play the video, mute the video, unmute the video, rewind the video and play it again.
- Rate a video.
- Share a video with a friend.

This way, you are likely to uncover lots of problems you had not anticipated. For example, a screen reader user might not be able to find the search box or the controls for the video. Conversely, users might have navigational strategies for dealing with the web you do not even know about.

Interpreting the results

In an ideal world, we could test every possible combination and get feedback from everybody. But in reality, time and money will limit user testing. Given this, feedback can be a double-edged sword. While it can teach you an enormous amount, there is a real danger of attaching too much weight to one person's view, which may not be representative of the greater target audience. For example, some screen reader users tend to be looking for an experience streamlined for a blind users; others are keen to know everything about the site that their sighted friends and colleagues see.

This is where accessibility standards like WCAG really come into their own. By following such guidelines, you can increase your chances of getting a foundation of accessibility even for user groups you are not able to test.

When you do observe a problem, analyse its causes. For example, your video sharing site includes a page showing popular videos in a data table, with columns including a still, a title, uploaded date, last played date, and overall rating, and arranged in row groups by category of video. In user testing, a screen reader user has trouble using the data table. This could reflect:

- A problem with the site code. For example, maybe the developers constructed a data table from meaningless `div` elements rather than using proper data table markup. Here the appropriate action would be to recode the table.
- Inexpertise on the part of the user. For example, a JAWS user might be unfamiliar with JAWS's features for navigating and reading data tables. Here an appropriate action might be to provide additional documentation or hints for less expert users. If expert users do not make ideal test subjects, they make great consultants on points such as this.

-
- A problem with the user agent. For example, Safari exposes data tables to the Apple accessibility model as a series of layout boxes rather than as a set of data relationships. Here appropriate actions would include reporting the bug to the user agent vendor or developers, researching a technique that does work in the user agent, or noting the limitation in documentation and suggesting alternative user agents that do work with your web site.
 - A problem with the screen reader. For example, the developers might have shortened long table headers using the `abbr` attribute, but the screen reader might not provide a user interface for reading the shortened version. Here appropriate actions would include reporting the bug to the screen reader vendor or developers, and might be to find a technique that does work in the screen reader, or to note the limitation in documentation and suggest an alternative tool or navigation strategy that does work.

Communicating the results of accessibility testing

When communicating the results of accessibility evaluation, document precisely what was evaluated. If you tested conformance with a particular standard, be specific about exactly where conformance has succeeded and failed. Whenever raising a problem, make sure to put it in real, human terms and explain how the problem might adversely affect users. Describe how to reproduce the problem and test for its resolution. Suggest practical techniques for achieving conformance or improving accessibility.

For example, you might report a problem with the video sharing website like this:

- *Problem: The dropdown menu cannot be opened without using a mouse to hover over top menu items, and the keyboard focus disappears off-screen as you tab through the menu.*
- *How to reproduce: Open the page in your browser and attempt to reach a subitem of the menu using the keyboard alone.*
- *Explanation: Web navigation should be device-independent, so that users using devices other than mice—such as blind users or users with motor disabilities—can access content and functionality. Currently, such users can not access the items in submenus and sighted users using the keyboard may be confused when the focus indicator disappears.*
- *Conformance implications: Keyboard operability is a requirement for WCAG 1.0 and WCAG 2.0 Level “A” compliance (see WCAG 1.0 Guideline 9 and WCAG 2.0 Guideline 2.1).*
- *Suggested remedies: When JavaScript is not available, use a simple list of links to subpages for each sublist of navigation. On sub pages, present the main navigation followed by the sublist. When JavaScript is available, remove the sublist from the DOM and add sublists for each menu item on the `click` event, which can be triggered by keyboards, mice, speech recognition, and touch screens alike.*

Summary

Not every webpage will receive an accessibility evaluation by experts and a suite of paid test subjects. But any web developer can learn the principles of accessibility, attempt to implement those principles in their code, and submit the results of their labours to user mailing lists to learn of further problems, and so feed new knowledge back into future development.

Exercise questions

- Try navigating a complex site of your choice without using the mouse. What difficulties do you encounter? How could the developers of the site help you?
- Turn off CSS and do your normal browsing for a day. What problems do you encounter?
- Turn off JavaScript and do your normal browsing for a day. What problems do you encounter?
- Pick a favourite site, design some personas for the site, then evaluate its conformance with WCAG 1.0 and general accessibility as an expert tester. Design a user testing plan for a site, and include recruit requirements and tasks to test. Write up a report on how it could improve its accessibility.

About the author



Original photo credit: [Ben Ward](#)

After studying a selection of medieval kings, eighteenth-century scientists, and other historical eccentrics at university, Benjamin Hawkes-Lewis somehow ended up working as a Web Developer at Yahoo!, much to his ongoing delight. His favourite things include a decent meal shared with friends, a good film in the cinema, lazing on the grass in the sunshine, and solving difficult problems by reference to primary sources, first principles, and empirical evidence.

27: CSS basics

BY [CHRISTIAN HEILMANN](#) · 26 SEP, 2008

Published in: [EXTERNAL](#), [SELECTORS](#), [RULES](#), [EMBEDDED](#), [COMMENTS](#)

This is Article 27 of the Web Standards Curriculum.

[Previous article—Accessibility testing](#)

[Next article—Inheritance and cascade](#)

[Table of contents](#)

Introduction

In the earlier tutorials of this course we talked about the content of web sites and how to structuring content using HTML. This is very important as it means that we give our documents meaning and structure for other technologies to tie into seamlessly. The most important web technology to discuss next is CSS (Cascading Style Sheets), which is used to style our HTML, and position it on the web page. In this article I'll introduce you to CSS—what it is, how to apply it to HTML, and what basic CSS syntax looks like. The structure of the article is as follows:

- [What is CSS?](#)
- [Defining style rules](#)
 - [CSS comments](#)
 - [Grouping selectors](#)
- [Advanced CSS selectors](#)
 - [Universal selectors](#)
 - [Attribute selectors](#)
 - [Child selectors](#)
 - [Descendent selectors](#)
 - [Adjacent sibling selectors](#)
 - [Pseudo-classes](#)
 - [Pseudo-elements](#)
- [CSS shorthand](#)
 - [Comparing individual and shorthand values](#)
 - [Providing less than four values for the margin property](#)
 - [Making the choice to use a single property or a shorthand value](#)
 - [Shorthand reference](#)
- [Applying CSS to HTML](#)
 - [Inline styles](#)
 - [Embedded styles](#)
 - [External style sheets](#)
 - [@importing stylesheets](#)
- [Summary](#)
- [Exercise questions](#)

What is CSS?

Whilst HTML structures the document and tells browsers what a certain element's function is (it is a link to another page? Is it a heading?), CSS gives the browser instructions on how to display a certain element—styling, spacing and positioning. If HTML is the struts and bricks that make up the structure of a house, CSS is the plaster and paint to decorate it.

This is done using a system of rules, the exact syntax of which you'll learn more about below. These rules state what HTML elements should have styling added to them, and then within each rule list the properties (eg color, size, font, etc.) of those HTML elements they want to manipulate, and what values they want to

change them to. For example, a CSS rule might state “I want to find every `h2` element and colour them all green”, or “I want to find every paragraph with a class name of `author-name`, colour their backgrounds in red, make the text inside them twice the size of normal paragraph text, and add 10 pixels of spacing around each one.

CSS is not a programming language like JavaScript and it is not a markup language like HTML—actually there is nothing that can be compared to it. Technologies that defined interfaces before web development always mixed presentation and structure. This is not a clever thing to do in an environment that changes as often as the web, which is why CSS was invented.

Defining style rules

Without further ado, let's have a look at a CSS code example, and then dissect it:

```
selector {  
  property1:value;  
  property2:value;  
  property3:value;  
}
```

The pertinent parts are as follows:

- The selector identifies the HTML elements that the rule will be applied to, identified by the actual element name, eg `body`, or by other means such as `class` attribute values—we'll get on to these later.
- The curly braces contain the property/value pairs, which are separated from each other by semicolons; the properties are separated from their respective values by colons,
- The properties define what you want to do to the element(s) you have selected. These come in wide varieties, which can affect element color, background color, position, margins, padding, font type, and many other things.
- The values are the values that you want to change each property of the selected elements to. The values are dependent on the property, for example properties that affect color can take hexadecimal colours, like `#336699`, RGB values like `rgb(12,134,22)` or colour names like red, green or blue. Properties that affect position, margins, width, height etc can be measured in pixels, ems, percentages, centimeters or other such units.

Now let's look at a specific example:

```
p {  
  margin:5px;  
  font-family:arial;  
  color:blue;  
}
```

The HTML element this rule selects is `p`—every `p` in the HTML document(s) that use this CSS will have this rule applied to it, unless they have more specific rules also applied to them, in which case the more specific rule(s) will overwrite this rule. The properties affected by this rule are the margins around the paragraphs, the font of the text inside the paragraphs, and the color of that text. The margins are set at 5 pixels, the font is set as Arial, and the color of the text is set as blue.

We will come back to all of these specifics later—the main goal of this tutorial is to cover the basics of CSS and not the nitty-gritty details.

A whole set of these rules goes together to form a style sheet. This is the most basic syntax of CSS there is. There is more, but not much—probably the coolest thing about CSS is its simplicity.

CSS comments

One thing to know early on is how to comment in CSS. You add comments by enclosing them in `/*` and `*/`. Comments can span several lines, and the browser will ignore these lines:

```
/* These are basic element selectors */
selector{
  property1:value;
  property2:value;
  property3:value;
}
```

You can add comments either between rules or inside the property block—for example in the following CSS the 2nd and 3rd properties are enclosed inside comment delimiters, so they will be ignored by the browser. This is useful when you are checking out what effect certain parts of your CSS is having on your web page; just comment them out, save your CSS, and reload the HTML.

```
selector{
  property1:value;
  /*
  property2:value;
  property3:value;
  */
}
```

Unlike other languages, CSS only has block level comments—single line comments do not exist. You can of course constrain the comment to a single line if you wish, but you still need to include the opening and closing comment delimiters (`/*` and `*/`).

Grouping selectors

You can also group different selectors. Say you want to apply the same style to `h1` and `p`—you could write the following CSS:

```
h1 {color:red}
p {color:red}
```

This however is not ideal, as you repeat information that is the same. Therefore you can shorten the CSS by grouping the selectors together with a comma—the rules within the brackets are applied to both selectors:

```
h1, p {color:red}
```

There are several different selectors, each matching a different part of the markup. The three most basic ones that you'll encounter most often are as follows:

`example{ }`: element selector

matches all the elements of the name *example*, for example `p{ }` for all `<p></p>` elements

`.example{ }`: class selector

matches all elements that have a `class` attribute with the value of *example*, for example

`.warning{ }` for `<li class="warning">` and `<p class="warning"></p>`—note that class selectors don't test for any specific element name

`#example{ }`: id selector

matches the element with the `id` attribute of value *example* (for example `#nav{ }` for `<ul id="nav">`—note that ID selectors don't test for any element name, and you can only have one of each ID per HTML document—they are unique to each page.

You can see the above selectors in action in the following examples. Notice that when you open the example in a browser the `warning` style gets applied to both the list item and the paragraph.

- [example-selectors.html](#)
- [selectors.css](#)

You can join some selectors to define even more specific rules:

```
p.warning{ }
```

matches all paragraphs with the `class` of `warning`

```
div#example{ }
```

matches the element with the `id` attribute `example`, but only when it is a `div`

```
p.info, li.highlight{ }
```

matches paragraphs with a `class` of `info` and list items with a `class` of `highlight`

In the following example I use these to differentiate between the different warning styles:

- [example-specificselectors.html](#)
- [specificselectors.css](#)

Advanced CSS selectors

In the above section, I introduced you to the most basic of CSS selectors, element, class and id selectors. With these selectors you can accomplish a lot, but this certainly isn't the be all and end all of selectors—there are other selectors that allow you to select elements to style based on more specific criteria:

- Universal selectors: universal selectors can be used to select every element on the page.
- Attribute selectors: as their name suggests, attribute selectors allow you to select elements based on their attributes.
- Child selectors: if you want to select specific elements that are children of other specific elements, use this selector.
- Descendent selectors: if you want to select specific elements that are descendents of other specific elements (not just direct children, but further down in the tree as well), you can use this selector type.
- Adjacent sibling selectors: if you want to select just specific elements that follow other specific elements, use these selectors.
- Pseudo-classes: these allow you to style elements based not on what the elements are, but on more esoteric factors such as the states of links (eg if they are being hovered over, or have been visited already).
- Pseudo-elements: these allow you to style specific parts of elements, rather than the whole element (eg the first letter within that element); they also allow you to insert content before or after specific elements.

You will see references to some of the more complicated selectors as you progress through the rest of the curriculum, but don't worry if you don't understand them all immediately—you will get there as you gain more experience in styling web pages! It is best to start off easy with the three basic selectors mentioned in the above section, then move on to the others as you gain more confidence.

Universal selectors

Put simply, universal selectors select every element on a page to apply styles to. For example, the following rule says that every element on the page should be given a solid 1 pixel black border:

```
* {  
  border: 1px solid #000000;  
}
```

Attribute selectors

Attribute selectors allow you to select elements based on attributes they contain. For example, you can select every `img` element with an `alt` attribute with the following selector:

```
img[alt] {  
  border: 1px solid #000000;  
}
```

Note the square brackets.

Using the above selector, you could perhaps choose to put a black border around any images that have an `alt` attribute, and style other images with a bright red border—useful in accessibility testing.

But attributes instantly get more useful when you consider that you can select by *attribute value*, not just attribute names. The following rule gives all images with an `src` attribute value of `alert.gif`:

```
img[src="alert.gif"] {  
  border: 1px solid #000000;  
}
```

You might not think this is hugely useful, but again, it can be useful for debugging purposes. Far more useful is the ability to select for specific parts of attributes, for example file extensions. And this is on the way—CSS 3 actually introduces three new types of attribute selector that can select based on text strings in attribute values (at the beginning, end, or anywhere within the value). [Read Christopher Schmitt's article on CSS 3 attribute selectors](#).

Child selectors

You can use a child selector to select just specific elements that are children of other specific elements. For example, the following rule will turn the text of `strong` elements that are children of `h3` elements

blue, but no other `strong` elements: `p`

```
h3 > strong {  
  color: blue;  
}
```

Child selectors are not supported in IE 6 or below.

Descendent selectors

Descendent selectors are very similar to child selectors, except that child selectors only select direct descendents; descendent selectors select suitable elements anywhere in the element hierarchy, not just direct descendents. Let's look at what this means more carefully. Consider the following HTML snippet:

```
<div>
  <em>hello</em>
  <p>In this paragraph I will say
    <em>goodbye</em>.
  </p>
</div>
```

In this snippet, the `div` element is the parent of all the others. It has two children, an `em` and a `p`. The `p` element has a single child element, another `em`.

You could use a child selector to select just the `em` immediately inside the `div`, like so:

```
div > em {
  ...
}
```

If you instead used a descendent selector, as follows:

```
div em {
  ...
}
```

Both of the `em` elements would be selected.

Adjacent sibling selectors

These selectors allow you to select a specific element that comes directly after another specific element, on the same level in the element hierarchy. For example, if you wanted to select all `p` elements that come immediately after `h2` elements, but no other `p` elements, you could use the following rule:

```
h2 + p {
  ...
}
```

Adjacent sibling selectors are not supported in IE 6 or below.

Pseudo-classes

Pseudo-classes are used to provide styles not for elements, but for various states of elements. The most common usage you'll come across is styling link states, so I'll look at these first. The following list gives you the different pseudo-classes, and a description of the link state they select:

- `:link`—the normal, default state of links, just as you first found them.
- `:visited`—links that you have already visited in the browser you are currently using.
- `:focus`—links (or form fields, or anything else) that currently have the keyboard cursor within them.
- `:hover`—links that are currently being hovered over by the mouse pointer.
- `:active`—a link that is currently being clicked on.

The following CSS rules make it so that by default, links are blue (the default in most browsers anyway). When hovered over, the default link underline disappears. We want the same effect when the link is focussed via the keyboard, so we duplicate the `:hover` rule with `:focus`. When a link has already been visited, it turns grey. Finally, when a link is active, it is bolded, as an extra clue that something is about to happen.

```
a:link{
```

```
color: blue;
}
a:visited{
  color: gray;
}
a:hover{
  text-decoration: none;
}
a:active{
  font-weight: bold;
}
```

Take care if you don't specify these rules in the same order as they are shown in above, otherwise they might not work as you expect. This is due to the way specificity causes later rules in the stylesheet to override earlier rules. You'll [learn more about specificity in the next article](#).

The `:focus` pseudo-class is also useful as a usability aid in forms. For example, you can highlight the input field that has the active blinking cursor inside it with a rule like this:

```
input:focus {
  border: 2px solid black;
  background color: lightgray;
}
```

Next, I'll have a look at `:first-child`—this pseudo-class selects any instance of the element that is the first child element of its parent, so for example, the following rule selects the first list item (bulleted or numbered) in any list and makes its text bold:

```
li:first-child {
  font-weight: bold;
}
```

Lastly, I'll briefly mention the `:lang` pseudo-class, which selects elements whose languages have been set to the specified language using the `lang` attribute. For example, the following element:

```
<p lang="en-US">A paragraph of American text, gee whiz!</p>
```

Could be selected using the following:

```
p:lang(en-US) {
  ...
}
```

Pseudo-elements

Pseudo elements have two purposes. First of all, you can use them to select the first letter or first line of text inside a given element. To create a drop cap easily at the start of every paragraph of your document, you could use the following rule:

```
p:first-letter {  
    font-weight: bold;  
    font-size: 300%;  
    background-color: red;  
}
```

the first letter of every paragraph will now be bolded, 300% bigger than the rest of the paragraph, and have a red background.

To make the first line of every paragraph bold, you could use the following rule:

```
p:first-line {  
    font-weight: bold;  
}
```

The second use of pseudo-elements is generating content via CSS, which is more complicated. You can use the `:before` or `:after` pseudo-elements to specify that content should be inserted before or after the element you are selecting. You then specify *what it is* that you want to insert. As a simple example, you can use the following rule to insert a decorative images after every link on the page:

```
a:after{  
    content: " " url(flower.gif);  
}
```

You can also use the `attr()` function to insert the values of attributes of the elements after the element. For example, you could insert the target of every link in your document in brackets after them using the following:

```
a:after{  
    content: "(" attr(href) " )";  
}
```

Rules like this are great for print stylesheets, which are stylesheets you can write and which are automatically applied when a user prints a page. The advantage for the user is that you can hide all the navigation that a user can't follow in a printout, and use the technique above so the reader can see the URLs referenced on a page.

You can also insert incremented numerical values after repeating elements (such as bullets or paragraphs) using the `counter()` function—this is explained in much more detail in the [dev.opera.com article on CSS counters](http://dev.opera.com/article-on-CSS-counters).

These selectors are not supported in IE 6 or below. Also note that you shouldn't insert important information with CSS, or that content will be unavailable to assistive technologies or if a user chooses not to use your stylesheet. The golden rule is that CSS is for styling; HTML is for important content.

CSS shorthand

Another thing you'll come across regularly in this course is CSS shorthand. It is possible to combine several related CSS properties together into one property to save time and effort on your part. In this section I will look at the available types of shorthand.

I've already used shorthand in this section without mentioning it. The `border: 2px solid black;` rule is shorthand for separately specifying `border-width: 2px;`, `border-style: solid;` and `border-color: black;`.

Comparing individual and shorthand values

Consider the following margin rule (padding and border shorthand works in the same way):

```
div.foo {  
  margin-top: 1em;  
  margin-right: 1.5em;  
  margin-bottom: 2em;  
  margin-left: 2.5em;  
}
```

Such a rule could also be written as:

```
div.foo {  
  margin: 1em 1.5em 2em 2.5em;  
}
```

Providing less than four values for a shorthand property

A shorthand value can take less than four values according to the list below. The results are ordered by the number of values provided:

1. Same value applied to all four sides, for example `margin: 2px;`
 2. First value applied to the top and bottom, second to the left and right, for example `margin: 2px 5px;`
 3. First and third values applied to the top and bottom respectively, second value applied to the left and right, for example `margin: 2px 5px 1px;`
 4. Values applied to the top, right, bottom, and left respective to CSS source order, as seen above.
- Generally, the wisest course is to provide all four values to shorthand properties, for reasons of legibility. This advice also applies to the `padding` shorthand property.

Making the choice to use a single property or a shorthand value

Shorthand `margin` and `padding` properties tend to get the greatest share of use, though there are situations in which the shorthand properties are best avoided, or at least considered carefully, such as the following:

- Only a single margin needs to be set. In a situation where only one property needs to be set, the act of simultaneously setting multiple properties usually violates the KISS (Keep It Simple, Stupid) Principle, explained in the Glossary.
- The selector to which your properties apply is subject to many edge cases. When this happens—which it will, sooner or later—the inevitable heap of shorthand values can become hard to follow when it comes time to repair or alter your layout.
- The stylesheet you're writing will be maintained by people whose skills (or spatial reasoning ability) are deficient. If you can count on them to read this article you may not need to worry about this scenario, but it's best not to make any assumptions.
- You need to supplant a value, to account for an edge case. While this requirement is often a signal of a poorly designed document or stylesheet... those are hardly unheard of, either.

Shorthand reference

1. Border shorthand for different properties: Already explained at the very start of this section. One extra point to mention is that you can even set border properties values just for a single border of the element it is applied to like so:

```
2. border-left-width: 2px;  
3. border-left-style: solid;
```

```
border-left-color: black;
```

4. Margin, padding and border shorthand for same properties: These all act in the same way; shown as seen above in the [Comparing individual and shorthand values](#) section.
5. Font shorthand: You can specify the font size, weight, style, family and line height using one line shorthand. For example, consider the following CSS:

```
6. font-size: 1.5em;  
7. line-height: 200%;  
8. font-weight: bold;  
9. font-style: italic;
```

```
font-family: Georgia, "Times New Roman", serif;
```

You could specify all of this using the following line:

```
font: 1.5em/200% bold italic Georgia,"Times New Roman",serif;
```

10. Background shorthand: you can specify background color, background image, image repeat and image position with one line of CSS. Take the following:

```
11. background-color: #000;  
12. background-image: url(image.gif);  
13. background-repeat: no-repeat;
```

```
background-position: top left;
```

This can all be represented using the following shorthand:

```
background:#000 url(image.gif) no-repeat top left;
```

14. List shorthand: Again, a similar story with list properties allows you to put the property values for list bullet type, position and image on a single line. Take the following CSS:

```
15. list-style-type: circle;  
16. list-style-position: inside;
```

```
list-style-image: url(bullet.gif);
```

This is the equivalent of:

```
list-style: circle inside url(bullet.gif);
```

Applying CSS to HTML

There are three ways to apply CSS to an HTML document: inline styles, embedded styles and external style sheets. Unless you have a very good reason to use one of the first two always go for the third option. The reason for this will become obvious to you soon, but first review the different options.

Inline styles

You can apply styles to an element using a `style` attribute, like so:

```
<p style="background:blue; color:white; padding:5px;">Paragraph</p>
```

Inside this attribute you list all the CSS properties and their values (each property/value pair is separated from the others by a semi-colon, and each property is separated from its value within each pair by a colon.) This is how you define styles in CSS. [Try viewing the source of this example](#) (right/Ctrl + click > Source in Opera).

If you open this example in a browser you will see that the paragraph with the `style` attribute is blue with white text and has a different size to the others, as shown in Figure 1.

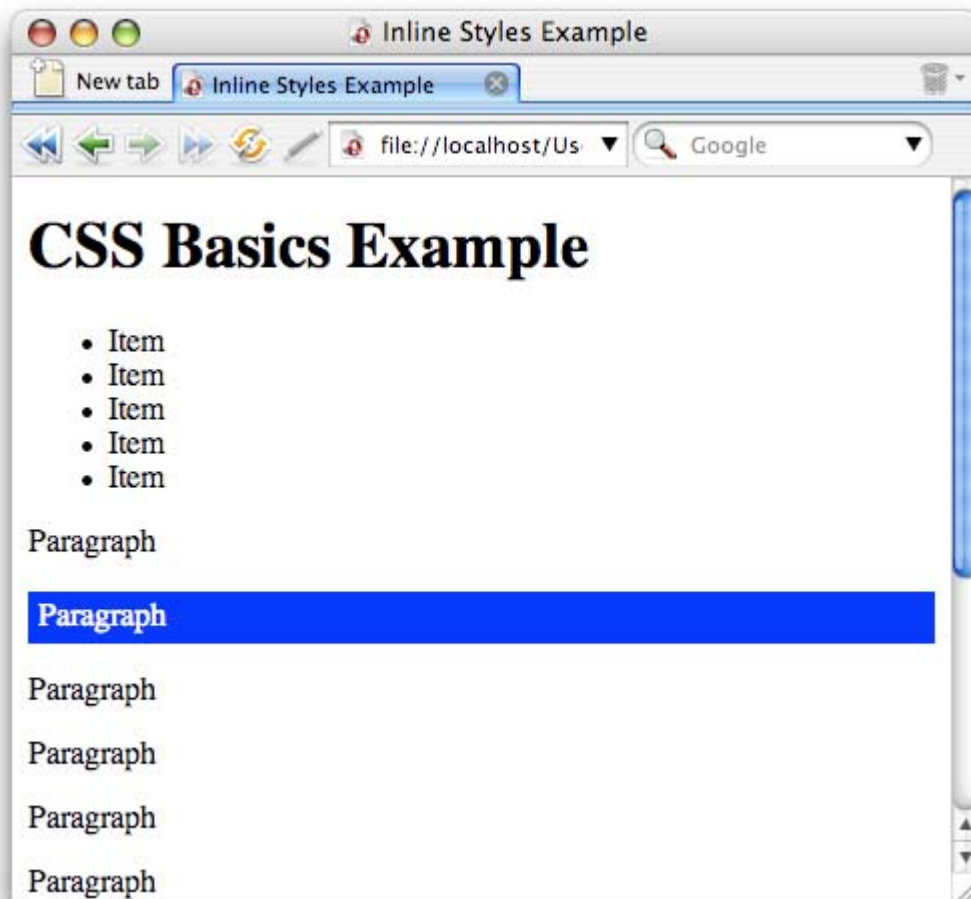


Figure 1: Opera shows the paragraph with the applied inline styles differently to the others.

The benefit of inline styles is that the browser will be forced to use these settings. Any other styles defined in other style sheets or even embedded in the head of the document will be overridden by these styles.

The big problem of inline styles is that they make maintenance a lot harder than it should be. Using CSS is all about separating the presentation of the document from the structure, but inline styles are doing just that—scattering presentation rules throughout the document.

In addition to the maintenance issue you don't take advantage of the most powerful part of CSS: the cascade. We'll come back to the cascade in detail in the next article, but for now all you need to know is that using the cascade means you define a look and feel in one place and the browser applies it to all the elements that match a certain rule.

Embedded styles

Embedded styles are placed in the head of the document inside a `style` element, [as in this example](#):

```
<style type="text/css" media="screen">
  p {
    color:white;
    background:blue;
    padding:5px;
  }
</style>
```

If you open the above link in a browser you'll see that the defined styles get applied to all the paragraphs in the document, as shown in Figure 2. Also try looking at the example page's source to see the CSS inside the head.

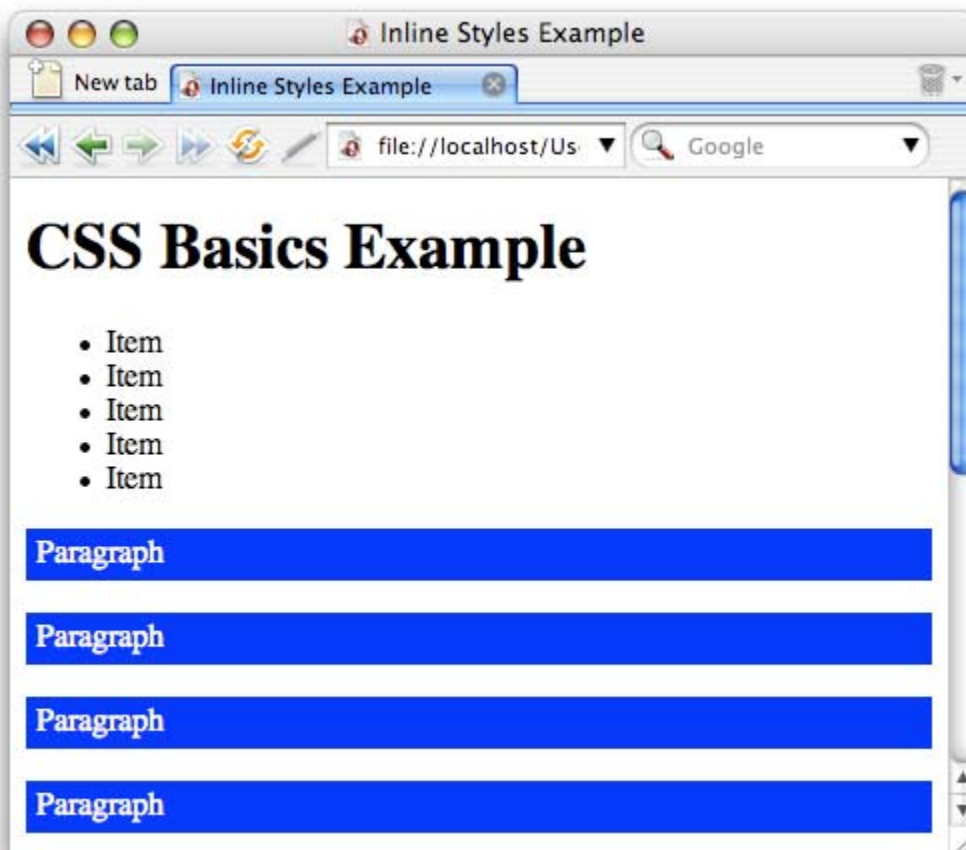


Figure 2: Opera shows all paragraphs with the styles defined in the embedded style sheet.

The benefit with embedded styles is that you don't need to add a `style` attribute to each paragraph—you can style them all with one single definition. This also means that if you need to change the look and feel of all paragraphs, you can do it in one single location, however this is still limited to one document—what if you want to define the look of paragraphs for a whole site in one go? Enter external style sheets.

External style sheets

External style sheets means putting all your CSS definitions in their own file, saving it with a file extension of CSS, and then applying it to your HTML documents using a `link` element inside the document head. View source of our [example page](#), and note that the head contains a `link` element that references this

[external CSS file](#), and verify that the styles defined in the external CSS file are applied to the HTML. Let's have a closer look at that `link` element:

```
<link rel="stylesheet" href="styles.css" type="text/css" media="screen">
```

We've talked about the `link` element before in this course. Just to recap: the `href` attribute points to the CSS file, the `media` attribute defines which media should get these styles applied to it (`screen` in this case as we don't want a printout to look like this) and the `type` defines what the linked resource is (a file extension is not enough to determine that).

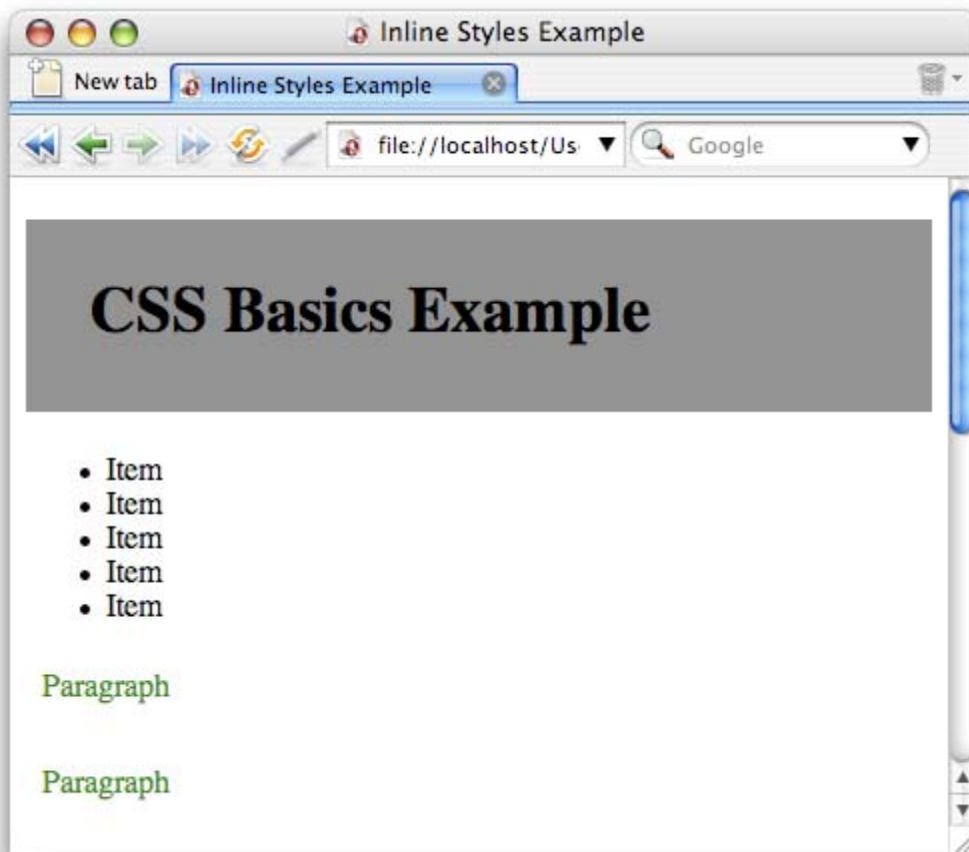


Figure 3: Opera shows the styles defined in the external style sheet when it is linked with a `link` element.

This is the best of all worlds: you keep your look and feel definitions all in one single file, which means that you can make site-wide changes by changing one file, and the browser can load that once and then cache it for all other documents that reference it, meaning less to download.

@importing stylesheets

There is actually another way to import external style sheets into HTML files - the `@import` property. This is inserted into an embedded style sheet, in the same way as the embedded CSS shown above. The syntax looks as follows:

```
<style type="text/css" media="screen">
  @import url("styles.css");

  ...other import statements or CSS styles could go here...
```

```
</style>
```

You'll sometimes see import statements without the brackets, but it does the same thing. Another thing to be aware of is that `@import` should always be first in an embedded style sheet. Finally, you can specify that the imported style sheet be applied only to certain types of media by including the media type at the end of the import statement (this works in every browser except IE6 and below). The following does the same thing as the previous code example:

```
<style type="text/css">
  @import url("styles.css") screen;

  ...other import statements or CSS styles could go here...
</style>
```

The first question you'll be asking is "why on earth do I need another way to apply external style sheets to my HTML documents?" Well, you don't really - I am mainly including information on `@import` here for the sake of completeness. There are a few minor advantages/disadvantages of using `@import` over `link` elements, but they are *very minor*, so it's really up to you which way you go. `link` elements are the recognised best way to do things these days.

- Older browsers don't recognise `@import` at all, so completely ignore it (Netscape 4 and older, and IE 4 and older if you omit the brackets from around the filename). You can therefore use an `@import` statement to hide styles from old buggy browsers that would use them incorrectly. You could put your up-to-date styles in an external stylesheet and import them with `@import`, then provide some really basic styles that will not cause IE/Netscape 4 to choke in the embedded stylesheet. This is useful, but you'll very rarely need to ensure IE/Netscape 4 compatibility these days!
- As mentioned before, IE6 doesn't support putting the media type at the end of the `@import` line, so they are not a good way to go if you want to insert multiple stylesheets for different media.
- You could argue that the code for multiple `@import` statements is smaller than the code for multiple `link` elements, but this is pretty negligible.

Summary

In this tutorial you learnt about applying CSS to HTML documents, either as inline styles using `style` attributes, embedded styles in a `style` element in the document head or as external files in their own document. You also learnt that the latter—linking an external style sheet using a `link` element—makes the most sense in terms of maintenance and caching. We then talked about the basic syntax of CSS and explained comments, different selector types, and grouping of selectors.

In the next tutorial we'll go deeper into the details of CSS, covering the cascade and specificity of selectors in detail.

Exercise Questions

- What are the benefits and problems of inline styles and how do you apply them to an element?
- What is a style rule? Describe the different components and syntax.
- Say you have a bunch of style rules, what do you need to do to make them into an external style sheet?
- What does the following CSS selector match: `ul#nav{ }?`
- What is the benefit of grouping selectors?
- How can you define a print style sheet?



About the author

Photo credit: [Bluesmoon](#)

Chris Heilmann has been a web developer for ten years, after dabbling in radio journalism. He works for Yahoo! in the UK as trainer and lead developer, and oversees the code quality on the front end for Europe and Asia.

Chris blogs at [Wait till I come](#) and is available on many a social network as “codepo8”.

28: Inheritance and Cascade

BY [TOMMY OLSSON](#) · 26 SEP, 2008

Introduction

Inheritance and the cascade are two fundamental concepts in CSS. Everyone who uses CSS needs to understand them. Fortunately, they aren't very difficult to grasp, although some of the details may be a bit hard to remember.

The two concepts are closely related, yet different. Inheritance is associated with how the elements in the HTML markup inherit properties from their parent (containing) elements and pass them on to their children, while the cascade has to do with the CSS declarations being applied to a document, and how conflicting rules do or don't override each other.

I will look closely at both concepts in this article. Inheritance is probably an easier concept to grasp, so I'll start with that, then progress to the intricacies of the cascade. [Download the source code for the examples](#) in this article; the zip contains the finished CSS and HTML, as well as the initial HTML template for you to work with as you go through the examples.

The contents of this article are as follows:

- [Inheritance](#)
 - [Why inheritance is useful](#)
 - [How inheritance works](#)
 - [An example of inheritance](#)
 - [Forcing inheritance](#)
- [The cascade](#)
 - [Importance](#)
 - [Specificity](#)
 - [Source order](#)
- [summary](#)
- [Exercise questions](#)

Inheritance

Inheritance in CSS is the mechanism through which certain properties are passed on from a parent element down to its children. It's quite similar to inheritance in genetics, really. If the parents have blue eyes, their children will probably have blue eyes, too.

Not all CSS properties are inherited, because it doesn't make sense for some of them to be. For instance, margins are not inherited, since it's unlikely that a child element should need the same margins as its parent. In most cases common sense will tell you which properties are inherited and which aren't, but to be really sure you need to look up each property in the [CSS 2.1 specification](#).

Why inheritance is useful

Why does CSS have an inheritance mechanism then? The easiest way to answer that is probably to consider what it'd be like if there was no such thing as inheritance. You would have to specify things like font family, font size and text colour individually—for every single element type.

Using inheritance, you can for example specify the font properties for the `html` or `body` elements and they will be inherited by all other elements. You can specify background and foreground colours for a specific container element and the foreground colour will automatically be inherited by any child elements in that container. The background colour isn't inherited, but the initial value for `background-color` is `transparent`, which means the parent's background will shine through. The effect is similar to what you'd get if the background colour were inherited, but consider what would happen if background *images* were inherited! Every child would have the same background image as its parent and the result would look

like a jigsaw puzzle put together by someone with a serious drug problem, since the background would “start over” for each element.

How inheritance works

Every element in an HTML document will inherit all inheritable properties from its parent *except* the root element (`html`), which doesn’t have a parent.

Whether or not the inherited properties will have any effect depends on other things, as you shall see later on when I talk about the cascade. Just as a blue-eyed mother can have a brown-eyed child if the father has brown eyes, inherited properties in CSS can be overridden.

An example of inheritance

1. Copy the following HTML document into a new file in your favourite text editor and save it as `inherit.html`.

```
2.    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">
3.    <html lang="en">
4.        <head>
5.            <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
6.            <title>Inheritance</title>
7.        </head>
8.        <body>
9.            <h1>Heading</h1>
10.           <p>A paragraph of text.</p>
11.        </body>
```

```
</html>
```

If you open the document in your web browser you will see a rather boring document displayed according to your browser’s default styling.

12. Create a new empty file in your text editor, copy the CSS rule below into it, and save the file as `style.css` in the same location as the HTML file.

```
13.    html {
14.        font: 75% Verdana,sans-serif;

    }
```

15. Link the style sheet to your HTML document by inserting the following line before the `</head>` tag.

```
<link rel="stylesheet" type="text/css" href="style.css">
```

16. Save the modified HTML file and reload the document in your browser. The font will change from the browser’s default (often Times or Times New Roman) to Verdana. If you don’t have Verdana installed on your computer, the text will be displayed in the default sans serif font specified in your browser settings.

The text will also become smaller; only three quarters of what it was in the unstyled version.

The CSS rule we specified applies only to the `html` element. We didn’t specify any rules for headings or paragraphs, yet all the text now displays in Verdana at 75% of the default size. Why? Because of inheritance.

The `font` property is a shorthand property that sets a whole number of font-related properties. We’ve only specified two of them—the font size and the font family—but that rule is equivalent to the following:

```
html {  
  font-style: normal;  
  font-variant: normal;  
  font-weight: normal;  
  font-size: 75%;  
  line-height: normal;  
  font-family: Verdana, sans-serif;  
}
```

All of those properties are inherited, so the `body` element will inherit them from the `html` element and then pass them on to its children—the heading and the paragraph.

But wait a second! There's something fishy going on with the inheritance of font size, isn't there? The `html` element's font size is set to 75%, but 75% of *what*? And shouldn't the font size of the `body` be 75% of its parent's font size, and the font sizes of the heading and the paragraph should be 75% of that of the `body` element, surely?

The value that is inherited is not the *specified value*—the value we write in our style sheet—but something called the *computed value*. The computed value is, in the case of font size, an absolute value measured in pixels. For the `html` element, which doesn't have a parent element from which to inherit, a percentage value for font size relates to the default font size set in the browser. Most contemporary browsers have a default font size setting of 16px. 75% of 16 is 12, so the computed value for the font size of the `html` element will probably be 12px. And *that* is the value that is inherited by `body` and passed on to the heading and the paragraph.

(The font size of the heading is larger, because the browser applies some built-in styling of its own. See the discussion about the cascade, below.)

17. Add two more declarations to the rule in your CSS style sheet:

```
18.  html {  
19.    font: 75% Verdana, sans-serif;  
20.    background-color: blue;  
21.    color: white;  
    }
```

22. Save the CSS file and reload the document in your browser.

Now the background is bright blue and all the text is white. The rule applies to the `html` element—the whole document—whose background will be blue. The white foreground colour is inherited by the `body` element and passed on to all children of `body`—in this case the heading and the paragraph. They don't inherit the background but it will default to `transparent`, so the net visual result will be white text on a blue background.

23. Add another new rule to the style sheet, and save and reload the document:

```
24.  h1 {  
25.    font-size: 300%;  
    }
```

This rule sets the font size for the heading. The percentage is applied to the inherited font size—75% of the browser default, which we have assumed to be 12px—so the heading size will be 300% of 12px, or 36px.

Forcing inheritance

You can force inheritance—even for properties that aren’t inherited by default—by using the `inherit` keyword. This should be used with care, however, since Microsoft Internet Explorer (up to and including version 7) doesn’t support this keyword.

The following rule will make all paragraphs inherit all background properties from their parents:

```
p {  
  background: inherit;  
}
```

With shorthand properties you can use `inherit` *instead of* the normal values. You have to use shorthand for everything or nothing—in longhand you can’t specify some values and use `inherit` for others, because the values can be given in any order and there is no way to specify *which* values we want to inherit.

Forcing inheritance isn’t something you need to do every day. It can be useful to “undo” a declaration in a conflicting rule, or to avoid hardcoding certain values. As an example, consider a typical navigation menu:

```
<ul id="nav">  
  <li><a href="/">Home</a></li>  
  <li><a href="/news/">News</a></li>  
  <li><a href="/products/">Products</a></li>  
  <li><a href="/services/">Services</a></li>  
  <li><a href="/about/">About Us</a></li>  
</ul>
```

To display this list of links as a horizontal menu, you could use the following CSS:

```
#nav {  
  background: blue;  
  color: white;  
  margin: 0;  
  padding: 0;  
}  
  
#nav li {  
  display: inline;  
  margin: 0;  
  padding: 0 0.5em;  
  border-right: 1px solid;  
}  
  
#nav li a {  
  color: inherit;  
  text-decoration: none;  
}
```

Here the background colour of the whole list is set to blue in the rule for `#nav`. This also sets the foreground colour to white, and this is inherited by each list item and each link. The rule for the list items sets a right border, but doesn’t specify the border colour, which means it will use the inherited foreground colour (white). For the links we’ve used `color:inherit` to force inheritance and override the browser’s default link colour.

Of course I could just as well have specified the border colour as white and the links’ text colour as white, but the beauty of letting inheritance do the job is that you now have only one place to change the colours if you decide to go with another colour scheme.

The cascade

CSS means Cascading Style Sheets, so it shouldn’t come as a surprise that the *cascade* is an important concept. It’s the mechanism that controls the end result when multiple, conflicting CSS declarations apply

to the same element. There are three main concepts that control the order in which CSS declarations are applied:

1. Importance
2. Specificity
3. Source order

I will look into these concepts below, one by one.

Importance is most ... er ... important. If two declarations have the same importance, the specificity of the rules decides which one will apply. If the rules have the same specificity, then source order controls the outcome.

Importance

The *importance* of a CSS declaration depends on *where* it is specified. The conflicting declarations will be applied in the following order; later ones will override earlier ones:

1. User agent style sheets
2. Normal declarations in user style sheets
3. Normal declarations in author style sheets
4. Important declarations in author style sheets
5. Important declarations in user style sheets

A *user agent style sheet* is the built-in style sheet of the browser. Every browser has its default rules for how to display various HTML elements if no style is specified by the user or designer of the page. For instance, unvisited links are usually blue and underlined.

A *user style sheet* is a style sheet that the *USER* has specified. Not all browsers support user style sheets, but they can be very useful, especially for users with certain types of disabilities. For instance, a dyslexic person can have a user style sheet that specifies certain fonts and colours that help reading.

Opera allows you to specify user stylesheets by going to Preferences... > Advanced tab > Content, clicking on the Style Options... button, then pointing to your user style sheet in the My style sheet text field inside the Display tab of this dialog box. You can also specify if you want the user style sheet to override the author (web designer's) stylesheet in the Presentation tab, and even add a button into the user interface that allows you to switch between the user and author style sheet. To do this, OK out of the Preferences... menu completely, then right- or Ctrl-click somewhere on the Opera browser interface, select Customize... > Buttons tab > Browser view, and drag the Author Mode button somewhere on to one of your toolbars.

An *author style sheet* is what we normally refer to when we say "style sheet". It's the style sheet that the author of the document (or, more likely, the site's designer) has written and linked to (or included).

Normal declarations are just that: normal declarations. The opposite is *important declarations*, which are declarations followed by an `!important` directive.

As you can see, important declarations in a user style sheet will trump everything else, which is logical. That dyslexic user might, for instance, want all text to be displayed in Comic Sans MS if he finds that font easier to read. He could then have a user style sheet containing the following rule:

```
* {  
  font-family: "Comic Sans MS" !important;  
}
```

In this case, no matter what the designer specified, and no matter what the browser's default font family is set to, everything will be displayed in Comic Sans MS.

The default browser rendering will only apply if those declarations aren't overridden by any rules in a user style sheet or an author style sheet, since the user agent style sheet has the lowest precedence.

To be honest, most designers don't have to think too much about importance, since there's nothing we can do about it. There is no way we could know if a user has a user style sheet defined that will override our CSS. If they do, they probably have a very good reason for doing so, anyway. Still, it's good to know what importance is and how it may affect the presentation of our documents.

Specificity

Specificity is something every CSS author needs to understand and to think about. It can be thought of as a measure of how specific a rule's selector is. A selector with low specificity may match many elements (like `*` which matches every element in the document), while a selector with high specificity might only match a single element on a page (like `#nav` that only matches the element with an `id` of `nav`).

The specificity of a selector can easily be calculated, as we shall soon see. If two or more declarations conflict for a given element, and all the declarations have the same importance, then the one in the rule with the most specific selector will "win".

Specificity has four components; let's call them a, b, c and d. Component "a" is the most distinguishing, "d" the least.

Component "a" is quite simple: it's 1 for a declaration in a `style` attribute, otherwise it's 0.

Component "b" is the number of `id` selectors in the selector (those that begin with a `#`).

Component "c" is the number of attribute selectors—including class selectors—and pseudo-classes.

Component "d" is the number of element types and pseudo-elements in the selector.

After a bit of counting, we can thus string those four components together to get the specificity for any rule. CSS declarations in a `style` attribute don't have a selector, so their specificity is always 1,0,0,0.

Let's look at a few examples—after this it should be quite clear how this works.

Selector	a	b	c	d	Specificity
<code>h1</code>	0	0	0	1	0,0,0,1
<code>.foo</code>	0	0	1	0	0,0,1,0
<code>#bar</code>	0	1	0	0	0,1,0,0
<code>html>head+body ul#nav *.home a:link</code>	0	1	2	5	0,1,2,5

Let's look at the last example in some more detail. You get `a=0` since it's a selector, not a declaration in a `style` attribute. There is one ID selector (`#nav`), so `b=1`. There is one attribute selector (`.home`) and one pseudo-class (`:link`), so `c=2`. There are five element types (`html`, `head`, `body`, `ul` and `a`), so `d=5`. The final specificity is thus 0,1,2,5.

It's worth noting that combinators (like `>`, `+` and the white space) do not affect a selector's specificity. The universal selector (`*`) has no input on specificity, either.

Also worth noting is that there is a huge difference in specificity between an `id` selector and an attribute selector that happens to refer to an `id` attribute. Although they match the same element, they have very different specificities. The specificity of `#nav` is 0,1,0,0 while the specificity of `[id="nav"]` is only 0,0,1,0.

Let's look at how this works in practice.

1. Start by adding another paragraph to your HTML document.


```
2.     <body>
3.     <h1>Heading</h1>
4.     <p>A paragraph of text.</p>
5.     <p>A second paragraph of text.</p>
```

```
</body>
```

6. Add a rule to your style sheet to make text in paragraphs have a different colour:

```
7.     p {
8.         color: cyan;
    }
```

9. Save both files and reload the document in your browser, and there should now be two paragraphs with cyan text.
10. Set an `id` on the first paragraph, so you can target it easily with a CSS selector.

```
11.    <body>
12.    <h1>Heading</h1>
13.    <p id="special">A paragraph of text.</p>
14.    <p>A second paragraph of text.</p>
```

```
</body>
```

15. Add a rule for the special paragraph in your style sheet:

```
16.    #special {
17.        background-color: red;
18.        color: yellow;
    }
```

19. Save both files and reload the document in your browser to see the now rather colourful result.

Let's look at the declarations that apply to your two paragraphs.

The first rule you added sets `color:cyan` for *all* paragraphs. The second rule sets a red background colour and sets `color:yellow` for the single element that has the `id` of `special`. Your first paragraph matches both of those rules; it is a paragraph and it has the `id` of `special`.

The red background isn't a problem, because it's only specified for `#special`. Both rules contain a declaration of the `color` property, though, which means there is a conflict. Both rules have the same importance—they are normal declarations in the author style sheet—so you have to look at the specificity of the two selectors.

The selector of the first rule is `p`, which has a specificity of 0,0,0,1 according to the rules above since it contains a single element type. The selector of the second rule is `#special`, which has a specificity of 0,1,0,0 since it consists of an `id` selector. 0,1,0,0 is much more specific than 0,0,0,1 so the `color:yellow` declaration wins and you get yellow text on a red background.

Source order

If two declarations affect the same element, have the same importance and the same specificity, the final distinguishing mark is the *source order*. The declaration that appears later in the style sheets will “win” over those that come before it.

If you have a single, external style sheet, then the declarations at the end of the file will override those that occur earlier in the file if there's a conflict. The conflicting declarations could also occur in different

style sheets. In that case, the order in which the style sheets are linked, included or imported controls which declaration will be applied. Let's look at a practical example of how this works.

1. Add a new rule to your style sheet, at the very end of the file, like so:

```
2. p {  
3.     background-color: yellow;  
4.     color: black;  
    }
```

5. Save and reload the web page. you now have *two* rules that match all paragraphs. They have the same importance and the same specificity (since the selector is the same), therefore the final mechanism for disambiguating will be the source order.

The last rule specifies `color:black` and that will override `color:cyan` from the earlier rule.

Note how the first paragraph isn't affected at all by this new rule. Although the new rule appears last, its selector has lower specificity than the one for `#special`. This shows clearly how specificity trumps source order.

Summary

Inheritance and the cascade are fundamental concepts that every web designer needs to understand.

Inheritance lets us declare properties on high-level elements and allows those properties to trickle down to all descendant elements. Only some properties are inherited by default, but inheritance can be forced with the `inherit` keyword.

The cascade sorts out all conflicts when multiple declarations would affect a given element. Important declarations will override less important ones. Among declarations with equal importance, the rule's specificity controls which one will apply. And, all else being equal, the source order makes the final distinction.

Exercise Questions

- Is the `width` property inherited? Think about it first—would it make sense?—then look up the correct answer in the [CSS specification](#).
- If we add `!important` to the `color:black` declaration in the last rule in our [example style sheet](#), will this have any effect on the text colour in the first, “special” paragraph?
- Which selector is more specific, “`#special`” or “`html>head+body>h1+p`”?
- What should a user style sheet look like to make our test document display in black Comic Sans MS on a white background, regardless of the author style sheet?

About the author



Tommy Olsson is a pragmatic evangelist for web standards and accessibility, who lives in the outback of central Sweden. He wrote his first HTML document in 1993 and is currently the technical webmaster for a Swedish government agency.

He has written one book so far—The Ultimate CSS Reference (with Paul O'Brien)—and has a sadly neglected blog called [The Autistic Cuckoo](#).

29: Text styling with CSS

BY BEN HENICK · 3 OCT, 2008

Introduction

Because the Web is a collection of *documents* – some dynamic, some static, some functional – the conventions under which they're formatted are borrowed from our best point of reference: six centuries of printing tradition. This includes typography. The Web however is a new and different medium, and web site typography needs a largely very different skill set to print design, and is subject to a lot more limitations.

This article builds upon the knowledge gained earlier on in the course, providing you with a detailed guide to styling text effectively using CSS. There are several examples linked to below, which will demonstrate the techniques discussed—here you can [download all the article 29 examples](#).

The article structure is as follows:

- [Web typography review](#)
 - [Contrast](#)
 - [Legibility and readability](#)
- [CSS font properties: changing the look of your text](#)
 - [font-size and unit type selection](#)
 - [How it's done](#)
 - [What unit types can be applied to CSS text properties?](#)
 - [What's the use of so many different unit types?](#)
 - [What is the *physical* equivalent of a desktop pixel?](#)
 - [Ems, percentages, and points, according to CSS](#)
 - [A brief note about the official CSS 2.1 Recommendation](#)
 - [Size keywords](#)
 - [font-family and typeface selection](#)
 - [code>font-stylecode, font-variant, and font-weight: changing the details](#)
 - [Why use the font-style property, when the em and i elements seem adequate?](#)
 - [font-variant as another tool for making short passages stand out](#)
 - [font-weight: boldness and the lack thereof](#)
 - [The font shorthand property](#)
- [CSS text and alignment properties: altering composition](#)
 - [text-align and justification](#)
 - [Proper justification of copy written in Western alphabets](#)
 - [Changing tracking: the letter-spacing and word-spacing properties](#)
 - [Using em units for good control](#)
 - [Indenting initial lines: the text-indent property](#)
 - [Capitalisation: the text-transform property](#)
 - [Link styling and showing deletions: the text-decoration propertycode](#)
 - [Borders, not underlines, with acronym and abbr](#)
 - [Leading adjustment and line-height](#)
 - [Supplanting pre and br: the white-space property](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

Web typography review

On the Web, designers have much less control over presentation than they do in other media. This is most obvious when it comes to document canvas properties such as size, resolution, and contrast. There are also severe limitations on the quality of Web typography, which were discussed in the [article about typography fundamentals](#).

Other subjects that deserve an introduction are *contrast*, *legibility*, and *readability*—I'll go through these now.

Contrast

Type contrast, the ease with which passages can be distinguished from whitespace and adjacent passages, is influenced by a number of factors such as luminosity, color, size, and composition. It is mentioned here for the sake of pointing out that low contrast copy should be set at the largest practicable size.

Legibility and readability

In a design context legibility is the ease with which a text passage can be *scanned* for specific pieces of information, while readability is the ease with which a passage can be *comprehended*. Design decisions that enhance one quality or the other are listed in Table 1.

Table 1: characteristics of legibility and readability				
Goal	Line length	Gutters (space between columns of text) & leading	Type choice	Justification
Readability	moderate	increased	serif	right-ragged [left]
Legibility	short	normal	sans-serif	variable, often full

Optimal column width will be discussed in the next article - [The CSS layout model](#).

CSS font properties: changing the look of your text

Typesetting involves the manipulation of text with respect to both composition and the look of individual letters and words. The latter class of tasks is handled by the CSS font properties, which will be discussed below.

font-size and unit type selection

Since documents typically vary type sizes more than typefaces, and variant fonts are usually handled well by user agent stylesheets, the primary property of interest is `font-size`. When used in a rule, it's followed with a value that specifies a unit measurement, or sometimes a keyword size (such as *small* or *medium*).

How it's done

The most important `font-size` declaration in a stylesheet looks something like this:

```
body { ...  
    font-size: 14px;  
    ... }
```

Inheritance causes all type size specifications in a document to be based upon the size declared for the document `body`, whether in the browser's stylesheet or in yours.

The typical browser default of 16 pixels is a good starting point for the size of your body copy, but most visitors can read smaller type with ease. As a result, many designers choose smaller sizes—around 11-14 pixels.

Inheritance applies to type sizing when a relative size is specified, and when a keyword size is specified for an element with a non-keyword-sized parent.

What unit types can be applied to CSS text properties?

In stylesheet rules, the unit types most commonly applied to text are pixels (px), ems (em, explained below), percentages (%), and points (pt). The behavior of text resized with these units is described in Table 2.

Table 2: convenient CSS units for text sizing

Unit	Definition ¹	Usage
CSS ems	$\Delta = x$	1.333em
Keywords	UA defined ²	large, &c.
Percentage	$\Delta = x \div 100$	133.3%
Pixels	absolute unit	16px
Points	absolute unit	12pt

¹ Δ is the ratio of change in type size from the inherited value.

² Only the nearest *non-keyword* size value is inherited.

Other available unit types include size keywords, picas (pc, one pica equals exactly 12 points), and exes (ex). Many of the other unit types supported by CSS2 are also available, but are rarely used when working with text.

What's the use of so many different unit types?

As pointed out in Table 2, there are *relative* and *absolute* sizing units. Keywords take on both characteristics – absolute sizing with respect to one another, but relative to the non-keyword value they inherit. The best practices to follow for their use are as follows:

- Absolute sizes (px, standardized units such as pt) are best used in layouts that do not change in relation to document canvas properties – so-called “fixed,” “static,” or “Ice” layouts.
- Relative sizes (em, %) should be used in non-static layouts, and in situations where a compromise needs to be struck between site usability and the designer's control over the layout.
- Keyword sizes (explained below) should be used when usability trumps all other design considerations.

Absolute sizes and usability

Older versions of Internet Explorer do not allow the visitor to resize text that has been set at absolute sizes, and the text resizing interfaces of the browsers that *do* allow that degree of user control tend to be hard-to-find. Because of this constraint, it's a common practice to set the `font-size` of `body` to a relative value, usually in em units that are assumed to be controlled by the browser default.

What is the *physical* equivalent of a desktop pixel?

The most accurate answer to this question is that there is no such thing. Pixels are a unit of display hardware resolution, and nothing more. However...

Despite the fact that it is strictly impossible to define or predict the literal size of a pixel, high-strung commercial project sponsors tend to be unpleasantly surprised when they discover that the design of their site takes on a different look on client hosts that are configured differently to their own, and yell at the web designer because of this. For that reason, it can be helpful to understand how pixels behave—I'm

giving you this as ammunition, ready for those times when anyone you are creating a web site for complains that the text doesn't look exactly the same on different machines.

Software publishers have an informal understanding that 96 ppi (pixels per inch) is a reasonable measurement. Thus 16 pixel body copy will print at a size of one-sixth of an inch, or 12 points. On the increasingly typical 17" 1280x800 liquid crystal display, such 16 pixel copy will have an approximate size of 13 points on screen, but on a similar 15" notebook display, its size will be 11.44 points.

These measurements are effective at default settings. Most environments allow the end user to set a custom ppi measurement, so edge cases will arise.

In conclusion: a pixel is a pixel, but everything else is variable.

Ems, percentages, and points, according to CSS

Traditionally, the *em* is equivalent to the height of an uppercase "M" in the font to which it applies. However, in CSS the `em` unit is actually equivalent to the total height of one line; in other words for an element that has had its `font-size` set to 14 pixels:

`1em = 100% = 14px`

In typical environments, this statement above can be expanded to:

`1em = 100% = 14px = 10.5pt`

Increasing or decreasing sizes work multiplicatively, so if you have a second element that you want to set to a size of 16 pixels, given normal inheritance all of the following would obtain the desired result:

`1.143em = 114.3% • 16px = 12pt`

A brief note about the official CSS 2.1 Recommendation

You will find yourself directed on occasion to consult the World Wide Web Consortium's [Candidate Recommendation for the CSS 2.1 Specification](#). Like all W3C Recommendations, this document can be considered the definition of a Web standard; some are followed more faithfully than others, by browser manufacturers as well as web developers.

While knowledge of W3C Recommendations in both breadth and depth is good to have, this course has been written to give you a concise but easy to digest introduction to web development/design, and the W3C recommendations can be a bit verbose, to say the least. All cases here in which you are directed to visit the CSS 2.1 Recommendation point to material that's too obscure to justify accurate explanation in this article.

Size keywords

You can also use size keywords, as briefly mentioned above. There are seven of them, from `xx-small` up to `xx-large`, with `medium` being the middle (and default) value. The full list of all seven values is spelled out in Table 3 below, which includes all of the keywords supported by the various CSS properties discussed in this article.

The CSS 2.1 Recommendation provides a wealth of additional detail about `font-size` [keywords](#).

Demonstration 1

From time to time, this text will link to a progressively styled demonstration document that will display the CSS properties under discussion, in actual use. This first instance shows the example document HTML completely unstyled

Links:

- [Unstyled demonstration document](#)

- [Resized title, attributions, and bodycopy](#)
- [Demo 1 stylesheet](#)

New rules

```
body { font-size: 14px; }
h1 { font-size: x-large; }
.sectionNote { font-size: medium; }
.attribution { font-size: small; }
```

font-family and typeface selection

With text sized to your satisfaction, you can move onto selecting a typeface or two. This is accomplished with the `font-family` property, which is used as shown in Demonstration 2 below.

When providing a value for the `font-family` property, you should follow these rules:

1. Faces must be named exactly as they are named in the font library of the client host, using the non-variant font as a guide.
2. All named faces must be separated by commas, with or without trailing whitespace.
3. In cases where the name of a face contains more than one word, it must be enclosed by single or double quotes. Example: 'Times New Roman'.
4. Faces should be named in order of *ascending* likelihood of availability. For example, if you want Macintosh users to see a page set in Palatino, your property-value declaration should probably read as follows: `font-family: Palatino, Georgia, serif;`. Palatino is exactly what you want, but some users may not have it; Georgia is much more likely to be available, and is similar to Palatino; serif refers to a generic default serif font—if neither Palatino nor Georgia are available, the system will fall back to its default serif font.
5. As a fail-safe, a `font-family` value should always end with the appropriate generic name, as explained above. The typefaces used in generic families by MacOS 10.5 are displayed in Figure 1.

<i>Excitably</i>	cursive: Apple Chancery
Excitably	fantasy: Papyrus
Excitably	monospace: Monaco
Excitably	sans-serif: Helvetica
Excitably	serif: Times New Roman

Figure 1: Default “generic” typefaces in MacOS 10.5, as rendered at 24px by Safari 3.1.

The CSS 2.1 Recommendation lists [several more typefaces that might apply to each generic family](#).

Demonstration 2

Now that the size of the text is predictable, we want to optimize the typefaces used. The title is best set in a sans serif face for legibility, and the narrative itself in a serif face.

Links:

- [New fonts](#)
- [Demo 2 stylesheet](#)

New rules

```
body { font-family: Palatino, 'Palatino Linotype', Georgia,
```

```
        sans-serif; }  
h1 { font-family: Helvetica,Arial,sans-serif; }  
blockquote { font-family: Helvetica,Arial,sans-serif; }
```

font-style, font-variant, and font-weight: changing the details

The `font-style` property manipulates italicization without resorting to the `i` element; its three valid values are `italic`, `oblique`, and `normal`.

The `italic` and `oblique` values deliver functionally identical results in the most recent versions of all mass-market Web browsers, even though there is a meaningful *technical* difference between the two styles, as explained in the [Typography Glossary](#) that accompanies this article.

Why use the `font-style` property, when the `em` and `i` elements seem adequate?

Each of those elements has specific uses: the `em` to provide emphasis, and `i` to serve as an alternative to `...` in those few instances when its use would be appropriate. Generally `<i>` isn't appropriate at all, as it is a presentational element, but there are certain pieces of copy that are italicized by convention, such as book titles (although this is still arguable; some think the `cite` element is more appropriate for book titles. Do what seems most appropriate). `` is generally more appropriate, because it specifies emphasis as a concept, rather than italics as a specific style—the actual look of emphasis may vary between different browsers.

There are situations where the use of `em` and its cousin `strong` may require a different approach. For example, suppose you wanted to emphasize copy by enlarging it. The consistent step for providing strong emphasis would then be to add italicization, resulting in rules such as the following:

```
em {  
    font-size: large;  
    font-style: normal;  
}  
  
strong {  
    font-size: large;  
    font-weight: normal;  
    font-style: italic;  
}
```

The preceding stylesheet rules would deliver results similar to the following:

The *quick* red fox jumped over the lazy brown dog.

Demonstration 3

There are no italicized words or passages in the demonstration copy, and the attributions already contain the necessary italicization due to use of the `cite` element. Given the lack of options, the title is the best candidate for italicization.

Links:

- [Italicize the title](#)
- [Demo 3 stylesheet](#)

New rules

```
h1 { font-style: italic; }  
.sectionNote { font-style: normal; }
```

font-variant as another tool for making short passages stand out

The `font-variant` property has two possible values, `small-caps` and `normal`. “Small caps” (also known as “copperplate” letters) are used by some designers to highlight the opening phrase of a long narrative, or to provide emphasis for quoted signage, such as the following:

ABANDON ALL HOPE, YE WHO ENTER HERE.

Demonstration 4*Links:*

- [Set the opening phrase in small caps](#)
- [Demo 4 stylesheet](#)

New rules:

```
.opening { font-variant: small-caps; }
```

font-weight: boldness and the lack thereof

The `font-weight` property allows you to alter the level of boldness of any passage of text to which it is applied.

The commonly supported values of the `font-weight` property are `normal` and `bold`. While the CSS 2.1 Recommendation provides [several other values](#), the current state of Web typography support makes those other values functionally meaningless to all but specialist users.

Demonstration 5

Boldfacing the name of an author is a design technique more often used in periodicals, but it still fits in various situations on the web.

Links:

- [Boldface the author's name](#)
- [Demo 5 stylesheet](#)

New rules:

```
.author { font-weight: bold; }
```

The `font` shorthand property

Many text properties can be provided in the value for a single property, if needed, in a manner similar in nature to background properties.

A font shorthand rule looks like this:

```
h1 { font: italic normal bold x-large/1.167em Helvetica,Arial,sans-serif; }
```

For the best results, the value you supply for this property should include your intended values for all of the following individual properties in the following order, separated by spaces:

1. `font-style`
2. `font-variant`
3. `font-weight`
4. `font-size`, followed if necessary by a slash and the value of `line-height` (see below)
5. `font-family`, which in this instance can also be a reserved word specifying a system font;
multiple values should be separated by commas but not spaces

Table 3: supported keyword values for properties discussed in this article

property	Values
font-family	cursive, fantasy, monospace, sans-serif, serif
font-size	xx-small, x-small, small, medium, large, x-large, xx-large
font-style	italic, normal, oblique
font-variant	normal, small-caps
font-weight	bold, normal
line-height	normal
text-align	center, justify, left, right
text-decoration	line-through, none, overline, underline
text-transform	capitalize, lowercase, none, uppercase
white-space	normal, nowrap, pre, pre-line, pre-wrap

CSS text and alignment properties: altering composition

A stylist working with letters and words is dealing in details: lines, curves, dots, pixels, and the other *cellular* bits of a design. A design is a whole thing however; it has bigger components which are brought into focus by composition controlled principally through the CSS layout model. In addition to that layout model, CSS also implements text and alignment properties that influence composition. The rest of this article will discuss those properties.

text-align and justification

As is the case with word processing environments, the `text-align` property supports four justification values: `left`, `right`, `center`, and `justify`. The last of these provides *full* justification: visible text margins that are consistent on both the left and right sides of a block of copy.

Proper justification of copy written in Western alphabets

The best general usage of the different available alignments is as follows:

- Left justification is best suited to long passages of narrative.
- Right justification is best used on the leftmost column of data tables and multiple column layouts. When the adjacent column is then left justified and placed on the other side of an appropriately wide gutter, the result is to improve the legibility of both columns.
- Full justification works well for small blocks such as block quotations and teaser copy. When used on long passages of narrative set to optimal width bestride wide margins, full justification also improves the apparent coherence of your layout.
- Centering is typically used for interface elements and serial lists such as those found in site footers.

Demonstration 6

Since the demonstration is built around fiction originally sourced from a book, why not give it full justification?

Links:

- [Apply full justification to the bodycopy of the passage](#)
- [Demo 6 stylesheet](#)

New rules:

```
p { text-align: justify; }
blockquote p { text-align: left; }
```

Changing tracking: the `letter-spacing` and `word-spacing` properties

These properties are fairly self-explanatory; they allow you to alter the amount of whitespace between letters and words, respectively.

The most common use of `letter-spacing` is to provide subtle emphasis analogous in effect to that provided by `font-variant: small-caps`; it may also be used to subtly alter the composition of interface elements.

The `word-spacing` property is best used to deliberately change the number of words likely to appear on a single line of copy. For example, it might be used if you have a copy block of typical width, but atypical type size.

In print, letterspacing and word spacing are also applied on an ad hoc basis to avoid composition flaws, but on the Web this technique has limited usefulness.

In addition to unit values, these properties support the `normal` value.

Using em units for good control

When you change the letterspacing of text, a little nudge carries for a long distance; the default letterspacing tends to be between one-tenth and one-twentieth of an em, so `letter-spacing` values that do more than double or halve the default might well result in illegible copy.

Demonstration 7

The sign copy discussed near the end could use some subtle emphasis... and since the pull quote is set in a larger size, it can be improved using some word spacing.

Links:

- [Add letterspacing to the proposed sign greetings in the penultimate paragraph of the passage](#)
- [Demo 7 stylesheet](#)

New rules:

```
q { letter-spacing: .143em; }
.pullQuote { word-spacing: .143em; }
```

Indenting initial lines: the `text-indent` property

The `text-indent` property makes it possible to indent paragraphs in the traditional style of printed matter. There are also a number of advanced layout techniques that are made possible by this property and its support of negative values.

The values supported by this property are the same as those supported by `font-size`, with the addition of `normal`.

Demonstration 8

On the same rationale that the passage was fully justified, maybe it should have all of its paragraphs indented.

Links:

- [Provide initial indentation for the paragraphs of the bodycopy](#)
- [Demo 8 stylesheet](#)

New rules:

```
p { text-indent: 1.429em; }
blockquote p { text-indent: 0; }
```

Capitalisation: the `text-transform` property

Just as the `font-variant` property gives you access to copperplate letters, the `text-transform` property deals specifically with capitalization. Its allowed values provide for all-caps, all-lowercase, or all-initially-capitalized rendering of text. See Table 3 above for a list of supported values.

Demonstration 9

Links:

- [property the excerpt in all-caps](#)
- [Demo 9 stylesheet](#)

New rules:

```
.author { text-transform: uppercase; }
```

Link styling and showing deletions: the `text-decoration` property

This property makes it possible to put lines over, under, and through text. Its most common use is to manipulate default link underlines, however wikis, satire, and other settings will beg use of strikethroughs. In these cases you will want to employ the `ins` (insert) and `del` (delete) elements, which provide equivalent styling to the following:

```
ins {
  text-decoration: underline;
}

del {
  text-decoration: line-through;
}
```

Even without custom stylesheet rules, `ins` and `del` style markup as follows:

~~Mark Twain~~ Benjamin Disraeli is remembered for many witticisms, including “there are lies, damned lies, and statistics.”

Borders, not underlines, with `acronym` and `abbr`

Among some designers it is popular to alter the appearance of `acronym` and `abbr` elements so that they appear with what appears at first glance to be a *dotted* underline. However, this effect is actually accomplished with a `border-bottom` value.

Demonstration 10

Links:

- [Remove the underline from the link to the Gutenberg E-Text source for the demonstration copy](#)
- [Demo 10 stylesheet](#)

New rules:

```
.source a { text-decoration: none; }
```

Leading adjustment and `line-height`

It's well-known that placing some whitespace between lines of text tends to increase its readability, because the increased space ensures that ascenders and descenders (see Figure 2 for explanation) on adjacent lines will not compete for visual focus.

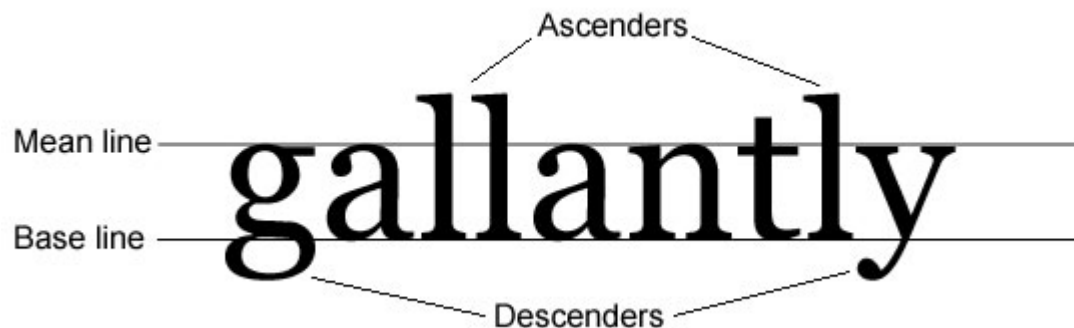


Figure 2: Ascenders are the parts of letters that rise above the mean line of the text; descenders are the parts of letters that drop below the base line that the text sits on.

There's a loose relationship between the `font-size` of an element and its `line-height`, but by default, all user agents insert a small amount of leading into each line of text – usually 10-15% of the height of the letters themselves. Because this default changes from one typeface to the next, the `line-height` property supports a value of `normal` in addition to numeric values.

It is also worth noting that unlike most CSS properties, `line-height` accepts numeric values without units, which are then rendered as a ratio of the default.

Demonstration 11

The relationship between leading and readability has been bandied about a lot, so here's the proof.

Links:

- [Apply a suitable amount of leading to the passage](#)
- [Demo 11 stylesheet](#)

New rules:

```
p { line-height: 1.5; }  
.attribution { line-height: 1.5; }
```

Supplanting `pre` and `br`: the `white-space` property

Forced linebreaks are a presentational element in the strictest sense, though there are many circumstances in which they're a desirable element of a site design. In tandem with browsers' habit of breaking lines on arbitrary spaces, exercising the desired degree of control with markup alone can be a challenge.

The `pre` element is provided to deal with these challenges, though it presents challenges of its own, which is why CSS offers the `white-space` property. Its supported values, which are listed in Table 3, allow the

stylist to choose whether or not the browser will render whitespace and linebreaks that have been added to source markup or inserted as generated content.

The CSS 2.1 Recommendation provides [exhaustive details about the suggested implementation and use of the white-space property](#).

Summary

An excellent site design requires a high level of attention to detail and proper adjustment of the interaction of numerous elements, not the least of which is type.

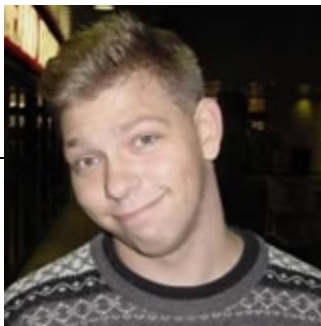
The suite of font and text properties supported by current browsers' implementations of CSS gives nearly the highest level of control over typesetting that existing output hardware will allow, and it's up to the conscientious site stylist to learn how to use them. As a result of using those properties successfully, sites go into production that can aspire to approach standards of quality in typesetting more commonly associated with the print medium we've developed over the span of centuries, in spite of the fact that the Web hasn't yet been around for longer than a single generation.

Bibliography

- Bos, Bert, et al. 2007. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. World Wide Web Consortium. <http://www.w3.org/TR/2007/CR-CSS21-20070719> etc. (accessed 28 May 2008).
- Chaparro, Barbara, et al. 2004. Reading online text: a comparison of four white space layouts. Wichita State University Software Usability Research Laboratory Usability News. <http://www.surl.org/usabilitynews/62/whitespace.asp> (accessed 28 May 2008).
- Dean, Paul. 2008. Extreme type terminology. I Love Typography, the Typography Blog. <http://ilovetypography.com/2008/03/21/extreme-type-terminology/> etc. (accessed 28 May 2008).
- Horton, Sarah, and Lynch, Patrick. 2002. [Web style guide](#): basic principles for creating web sites, 2nd edition. New Haven, Conn.: Yale University Press.
- Roselli, Adrian. 2002. A simple character entity chart. Evolt.org. http://www.evolt.org/article/A_Simple_Character_Entity_Chart/17/21234/ (accessed 28 May 2008).

Exercises

- List three HTML elements, other than `b` and `i`, which are typically rendered with variant fonts by default. Briefly describe the intended purpose of the elements you listed, and explain how the use of a variant font is appropriate to those elements' purposes.
- Subjectively test the readability of a long text passage of your choice, at varying `line-height` values. Briefly summarize your results.
- Apply `text-transform: uppercase;` to a single paragraph of the passage used for the previous exercise, and repeat your subjective test for readability. Briefly summarize your results.
- Briefly explain the advantages and disadvantages of anti-aliasing, using the [typography review](#) in this article for guidance.
- Describe the order in which typefaces should be specified, in a `font-family` value.
- In a closed-book setting, choose at least three properties described in this article and list at least one valid value (other than defaults) for each. Demonstrate or describe the results of the use of these property/value pairs in a stylesheet.
- Create an element, populate it with copy, and assign it a `font-size` value of `9px`. Open the document containing this element in Internet Explorer, and cycle through the type sizes provided in the View → Text Size menu. Evaluate the suitability of these results on sites with large numbers of middle aged and elderly visitors.



About the author

Ben Henick has been building Web sites in one capacity or another since September 1995, when he took on his first Web project as an academic volunteer. Since then, most of his work has been done on a freelance basis.

Ben is a generalist; his skillset touches on nearly every aspect of site design and development, from CSS and HTML, to design and copywriting, to PHP/MySQL and JavaScript/Ajax.

He lives in Lawrence, Kansas, with three computers and zero television sets. You can read more about him and his work at henick.net.

30: The CSS layout model - boxes, borders, margins, padding

BY [BEN HENICK](#) · 26 SEP, 2008

Introduction

At first glance, the CSS layout model is a straightforward affair. Boxes, borders, and margins are fairly simple objects, and CSS syntax provides a simple way to describe their characteristics.

However, browser rendering engines follow a long list of rules laid down in the CSS 2.1 Recommendation, and a few of their own. For this reason, there are a lot of details that need to be understood before advanced techniques can be added to a stylist's repertoire.

In this article you will be introduced to the CSS properties that manipulate the layout of HTML elements, including their borders, margins, and much more. Coverage will also include some of the rules mentioned above. Advanced column layout and grid-focused techniques will be discussed in later articles that will explore form layout, floats, clearing, and positioning in greater detail. There will be many code examples linked to throughout the article to demonstrate techniques discussed, but if you want to work through the code on your local machine, you can [download all the code examples here](#).

The contents of this article are as follows:

- [Changing composition: CSS margins, borders, and padding](#)
 - [Putting whitespace around an object: the margin-top, margin-right, margin-bottom, margin-left, and margin properties](#)
 - [Auto margins](#)
 - [Negative margins](#)
 - [Collapsing margins](#)
 - [Demonstration 1](#)
 - [Adding a border to an object: border properties](#)
 - [The border-width properties](#)
 - [The border-color properties](#)
 - [The border shorthand property and its four cousins, in more detail](#)
 - [Creating rules: the rationale for five border shorthand properties... instead of one](#)
 - [...And why so many properties? They're just borders, right?](#)
 - [Demonstration 2](#)
 - [When margins alone aren't enough: padding properties](#)
 - [Demonstration 3](#)
- [Working with element width and height](#)
 - [Width and height basics](#)
 - [min-width, max-width, min-height, and max-height](#)
 - [Demonstration 4](#)
 - [Overflow: fencing in code, or setting it free](#)
 - [The results of the four overflow values](#)
- [The CSS box models: fitting everything together](#)
 - [Choosing the right units for your layout](#)
 - [The principal rule of sizing elements: mix proportional and static units with care, or not at all](#)
 - [Choosing the right unit type for layout: advantages and disadvantages](#)
 - [The boxel model components](#)
 - [The W3C box model: everything is additive](#)
 - [Proportional margins and padding in the W3C box model](#)
- [Working with document flow](#)
 - [Element types and the display property](#)
 - [Demonstration 5](#)
 - [Causing elements to flow *around* others: the float property](#)
 - [Demonstration 6](#)

- [Forcing elements *below* their floated predecessors: the clear property](#)
- [Summary](#)
- [Further reading](#)
- [Exercise questions](#)

Changing composition: CSS margins, borders, and padding

Many HTML elements, such as `div` elements and headings, are rendered by default to occupy the entire width of the browser canvas and force a terminal linebreak, so that several such elements in series would render in a top-to-bottom stack on the document canvas.

However, HTML elements and the browser styles usually set for them are inadequate to the full range of use cases developers are called upon to consider in their work. The way CSS and HTML work together has been tuned to “fill in the gaps” so that `classes` and `ids` can add semantic meaning to markup while style sheet rules can precisely change the layout and presentation of content—perhaps even by canceling out large parts of the browser default styles altogether.

Careful control of whitespace is among a designer’s most important tools – and in the opinion of this author, the single most important. However, the degree of control over whitespace that brings high production values to a site design is absent from default browser stylesheets, which means that stylists typically make frequent use of the margin, border, padding, and other CSS layout properties explained in this article.

Margins, borders, and padding are arranged as shown in Figure 1.

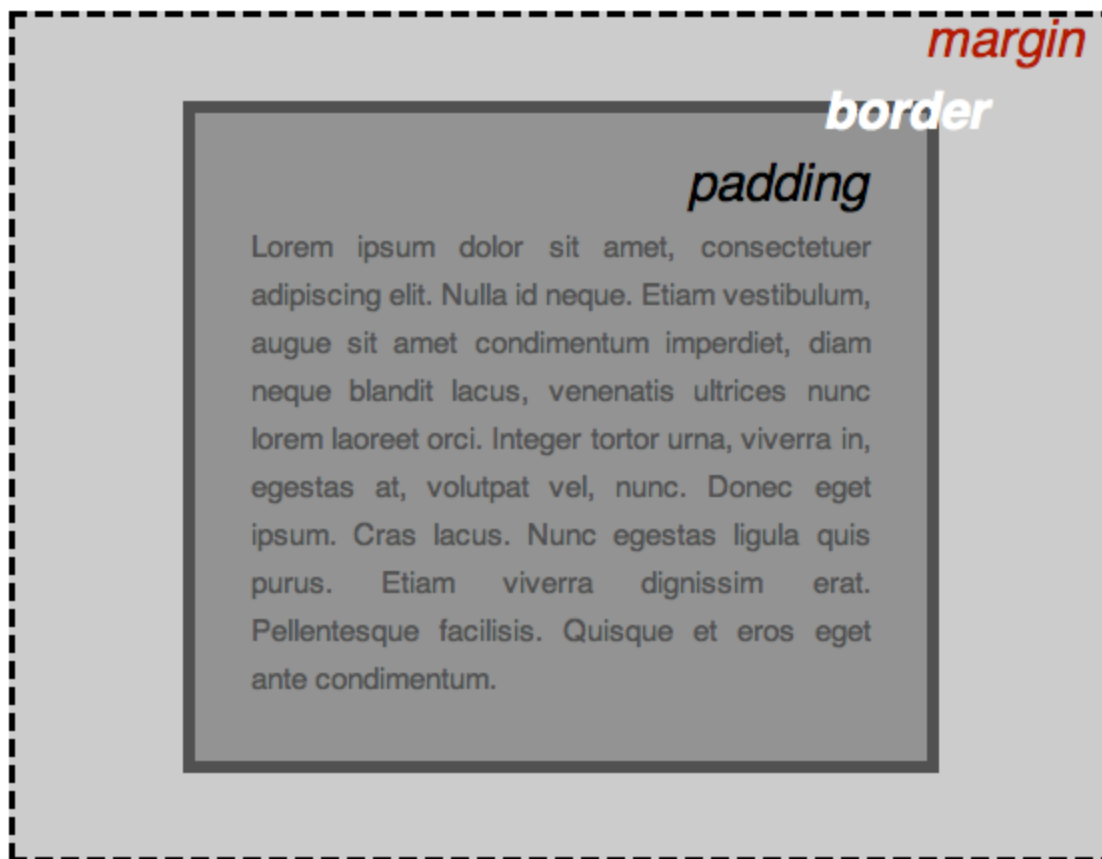


Figure 1: An explicit illustration of the various parts of an element box, labelled with associated CSS properties.

Putting whitespace around an object: the margin-top, margin-right, margin-bottom, margin-left, and margin properties

Margins can be specified singly, or in a shorthand rule. Furthermore, the shorthand rule still allows control of individual borders around an object. Valid values are usually specified in `px` or `em` units (pixels or ems). On print-specific stylesheets `in`, `cm`, or `pt` units might be used instead (inches, centimeters or points).

In all cases `%` (percentage) is a valid value, but needs to be used with care; such values are calculated as a proportion of the parent element's width, and careless provision of values might have unintended consequences. This challenge is explained in more detail during the discussion of the CSS box model below.

All inline elements except images lack margins, and will not take margin values. For a list of these elements, consult Table 2 below.

auto margins

Depending upon the circumstances, provision of an `auto` value instructs the browser to render a margin according to the value provided in its own stylesheet. However, when such a margin is applied to an element with a meaningful width, an `auto` margin instead causes all of the available space to be rendered as whitespace.

Given the following rule:

```
.narrowWaisted {  
  width: 16.667em;  
  margin: 1em auto 1em auto;  
}
```

...A block element of the class `narrowWaisted` will center itself in the middle of the available canvas.

...Or the right margin of an applicable element can be set to some relatively small value, while the left margin is assigned an `auto` value.

When that's done, such an element will instead appear nearly flush-right.

Negative margins

All of the margin properties can be assigned *negative* values. When this is done, an adjacent margin can be effectively "canceled out" to any degree. Given a large enough negative margin applied to a large enough element, the affected adjacent element can even be *overlapped*.

for example, consider the following simple `div` elements (taken from [example file negativemargin1.html](#).)

```
<div id="header"><h1>Lovely header</h1></div>  
<div id="content"><p>Overlapping text is entirely unreadable</p></div>
```

When styled with the following CSS

```
body {background-color:white font-family:Geneva, Arial, Helvetica, sans-serif;}  
#header {background-color:yellow}  
h1 {color:red font-size:2em; }
```

It creates the output shown in Figure 2:

Lovely header

Overlapping text is entirely unreadable

Figure 2: The two elements from our simple example. Nothing special to see here.

Here comes the interesting part. Now we'll add a fairly sizeable negative margin to the top of the bottom element, using the following rule:

```
#content {margin-top:-3em;}
```

This gives us the visual effect of shifting the element up so it overlaps with the heading, as shown in Figure 3 (see the [negativemargins2.html example file](#) for a live example).

Lovely header
Overlapping text is entirely unreadable

Figure 3: With a negative margin applied, the bottom element shifts upwards and overlaps the heading.

Collapsing margins

In cases where two similar and adjacent block elements share margins that are greater than zero, only the larger of the two margins will be applied. For example, take the following rule:

```
p {  
  margin: 1em auto 1.5em auto;  
}
```

If a document including this style rule is rendered *literally*, the resulting margin between two paragraphs in series would be 2.5em, as the sum of the bottom margin of paragraph 1 (1.5em) and the top margin of paragraph 2 (1em). However, due to the application of collapsing margins, the margin between them is only 1.5em.

Lists and headings are peculiar among block elements, so their margins will not be collapsed into the margins of the other block elements.

Demonstration 1

In the text styling article, the typesetting of the opening section of an F. Scott Fitzgerald story was done with many of the tools made available by CSS. For the demonstration in this article, that same page is being put to use again, with some minor changes (principally, the addition of a container element around all of the copy). The text styling is unchanged, but the few layout styles applied to that demonstration have been removed.

For starters, margins will be added to all of the elements that will need them.

Links:

- [Minimally styled demonstration document](#)

- [Beginning stylesheet](#)
- [New margins on body, title, pullquote, document container, and paragraphs](#)
- [Demo 1 stylesheet](#)

New rules:

```
body { margin: 0; }
#main { margin: 0 auto 0 auto; }
h1 { margin: 0 0 1em 0; }
.pullQuote { margin: auto 0 1em 1em; }
p { margin: 0; }
.attribution { margin: 0 0 1.5em 0; }
```

Adding a border to an object: border properties

There *is* a `border` shorthand property, but it's only useful when you want to provide a complete and consistent border around all four sides of an element. It's also possible to set the weight, style, and color of any of an element's four possible borders by using any meaningful combination of the following properties:

- `border-width`
- `border-style`
- `border-color`
- `border-top`
- `border-top-width`
- `border-top-style`
- `border-top-color`
- `border-right`
- `border-right-width`
- `border-right-style`
- `border-right-color`
- `border-bottom`
- `border-bottom-width`
- `border-bottom-style`
- `border-bottom-color`
- `border-left`
- `border-left-width`
- `border-left-style`
- `border-left-color`

The border-width properties

These properties behave exactly as one would expect: they assign explicit weight to one or more borders.

The `border-width` shorthand property accepts values in the same notation as the `margin` shorthand property, except that percentage values are unsupported. You might well see yourself writing a rule like the following:

```
td {
    border-width: 1px 0 0 1px;
}
```

The `border-style` properties

Figure 4: the eight common border styles in action.

The `border-style` properties commonly accept any of the following values:

`dashed`

The length of dashes, and the amount of whitespace between them, is determined by the browser.

`dotted`

The amount of whitespace between dots (which may take on any shape with an aspect ratio of 1) is determined by the browser.

`double`

The provided width will be divided into thirds and rendered in filled-negative-filled order.

`groove`

An `outset` will be rendered immediately inside and flush to an `inset`.

`inset`

The border will be shaded to make it appear as if the element to which it is applied is depressed into the canvas.

`none`

Equivalent to specifying a `-width` of zero.

`outset`

The border will be shaded to make it appear as if the element to which it is applied extrudes from the canvas.

`ridge`

An `inset` will be rendered immediately inside and flush to an `outset`.

`solid`

The border appears as an unbroken, unshaded line.

When the `border-style` shorthand property is used, it can accept up to four values which are applied in the same fashion as `margin` shorthand values.

The practice of obscuring a border (rather than omitting it) is handled by the `-color` properties.

The border-color properties

Finally, it's possible to set any color on any individual border, with either a single property such as those listed above, or the `border-color` shorthand property. Refer to the explanation of the `margin` shorthand property for details about the results of providing fewer than four values.

Like `background-color`, `border-color` can take a value of `transparent`. This can be useful in dealing with edge cases that require consistent composition but not consistent use of borders.

The border shorthand property and its four cousins, in more detail

Unlike the various `-width`, `-style`, and `-color` border properties, these five properties allow you to define the three characteristics of an object's four borders, or of any single border at a time. Valid `border` (etc) shorthand values contain any or all of the width, style, and color properties that apply to that border; the only limitation is that you must refer to either one side of an element at a time, or all four at once.

Consider the following `border` rule:

```
#borderShorthandExample {  
    border: 2px outset rgb(160,0,0);  
    padding: .857em;  
    background-color: rgb(255,224,224);  
}
```

An element to which the above rule is applied would look exactly like this paragraph.

When a value is omitted from a `border` shorthand rule, the rendered element will display a default result:

- Border width will be determined by the browser.
- Border style will be `solid`.
- Border color will be identical to the `color` applied to the element in question.

Creating rules: the rationale for five border shorthand properties... instead of one

The “rules” discussed here are lines drawn through a layout, not directives to follow. Such lines enhance contrast between an element and its neighboring space, and in many cases they help to create the illusion of depth within a layout. This last result is exemplified by the existence of the `inset` and `outset` border styles.

While these same effects can be accomplished by putting borders around all four sides of an element, the ability to draw precisely defined lines in a layout allows its designer considerable control over details.

...And why so many properties? They’re just borders, right?

When a layout is created which demands exceptional skill from a stylist, there will be a need to account for edge cases; this was already raised in the earlier discussion of margins.

Because of the way in which site designs are executed, you will encounter many cases where this element or that might have similar structural properties to other elements in a document, but have different presentation requirements. In these situations it makes perfect sense to write one rule for the most common case, and additional rules for each of the edge cases. It’s for this reason that the `auto` and `inherit` values exist: to use a default style as an edge case.

In the case of borders, edge cases might well require the alteration of a single characteristic of a border on a single side of an element – and when one wisely follows the KISS Principle, it’s usually best to stick to changing *only* those details which *need* to be changed.

Demonstration 2

Certain sections of the document should be given embellishment in the form of rules and borders.

Links:

- [Add a bottom rule to the title, and a border around the pullquote](#)
- [Demo 2 stylesheet](#)

New rules:

```
h1 { border-bottom: 1px solid rgb(153,153,153); }  
.pullQuote { border: 1px solid rgb(153,153,153); }
```

When margins alone aren’t enough: padding properties

You will encounter elements with background colors in secondary or accent hues that require gutters between content and margins. In other situations, you’ll need to provide space between borders and the copy near them.

In such cases and many others, you’ll get considerable use from the `padding`, `padding-top`, `padding-right`, `padding-bottom`, and `padding-left` properties. These properties insert negative space between the margins or borders of an element and its content. See Figure 1 above for a clear illustration of the relationship between margins, borders, and padding.

These properties behave in exactly the same manner as margin properties, with the following exceptions:

- `auto` values are functionally useless in references to padding properties.
- Negative padding values are invalid.
- Padding is never collapsed.
- Margin values are not applied to inline elements, but padding values are.

Demonstration 3

Gutters should be provided for the elements to which borders were previously added.

Links:

- [Insert gutters adjacent to the borders previously put on the title and pullquote](#)
- [Demo 3 stylesheet](#)

New rules:

```
body { padding: 0; }  
h1 { padding: .5em 0 .5em 0; }  
.pullQuote { padding: .5em; }
```

Working with element width and height

Most elements can have their dimensions altered as a matter of course. You've seen this capability demonstrated earlier in this article, during the discussion of `auto` margins.

The CSS properties used to manipulate the dimensions of elements are `width`, `height`, `min-width`, `max-width`, `min-height`, and `max-height`. These properties can then be divorced from (or linked to) the dimensions of element contents with the `overflow` property.

There's also a `clip` property which *hides* parts of an element *inside* its margins. However, it's omitted from this article because of its narrow scope of use.

width and height basics

As a rule, `width` and `height` produce exactly the results one would expect. However, their use carries some important caveats.

- `width` and `height` cannot be applied to `inline` elements... There are several elements (such as `span`, `strong`, and `em`) that will ignore the application of `width` and `height` values under typical circumstances. A list of these elements can be found in the discussion of element types, later in this article.
- ...except for images, which can be assigned `width` and `height` even though they are inline elements. The CSS 2.1 Recommendation refers to images as "replaced" elements, which means that the browsers should always treat them as possessing static dimensions. For this reason, those dimensions can be arbitrarily altered.
- `width` and `height` are only two of the properties that can influence the functional dimensions of an element. As a result, it's easy to put yourself in situations where an element is too small (usually too narrow) to hold its content as expected, leading to blowouts. The CSS box model discussion below addresses this issue.
- Rendering bugs in Microsoft Internet Explorer (IE) make it necessary to specify explicit `width` or `height` property/value pairs for some elements. There are some peculiarities about IE's rendering engine that can only be resolved with brute force (see the Glossary). Most of these peculiarities are known and slated to be removed from IE 8, but until that version has replaced its predecessors within the IE install base, this issue will be an inevitable test case. [PositionIsEverything.net](#) and the [CSS-Discuss Wiki](#) provide ample information about this issue and techniques that work around it.
- Rounding algorithms will, from time to time, cause off-spec differences in layout between browsers which display content via LCD, LED, or CRT (`type="screen"`) display media. The `screen` media type ultimately requires all units to be converted into pixel measurements, which may map differently from one browser to the next.

min-width, max-width, min-height, and max-height

From time to time you will encounter situations in which you need to constrain the size of an element – usually to ensure that a proportionally-sized column will always retain a readable width. The various `min-` and `max-` properties answer this requirement. As is the case with `width` and `height`, the results one can expect from using these properties are fairly predictable as a matter of course.

However, in the experience of this author, these properties have limited use (although other authors disagree). Like plain old `width` and `height`, they're subject to rounding errors that can deliver entirely unexpected results. More importantly, they are completely unsupported in IE 6, which still holds a considerable market share as of July 2008.

Demonstration 4

`auto` margins were placed to the left and right of the page container. Now it needs a `width` for those margin values to make any sense. Furthermore, the plan is to assign a `float` value to the `pullquote`, so that'll get a width, too.

Links:

- [Change the width of the document container and the pullquote](#)
- [Demo 4 stylesheet](#)

New rules:

```
#main { width: 42em; }
.pullQuote { width: 14em; }
```

overflow: fencing in content, or setting it free

When an element's width or height are set, it's sometimes necessary to consider what results are desired in the event that the contents of that element take up more space than is strictly available. This is especially true on sites with both user generated content and strict layout specifications.

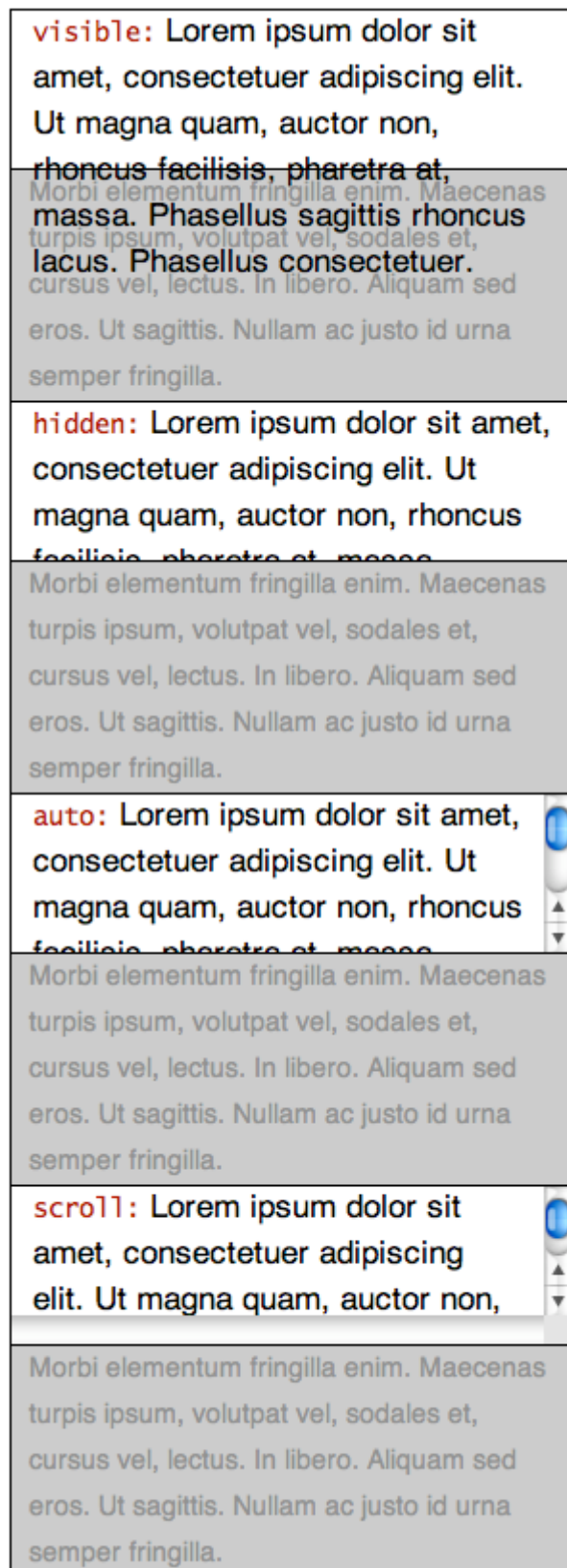
The `overflow` property and its four valid values – `visible`, `hidden`, `auto`, and `scroll` – are provided to handle such circumstances. Figure 5 illustrates the effect they have when applied to an element whose content spills out of its bounding box.

Figure 5: The effects of the CSS overflow property.

The results of the four overflow values

visible (default)

Contents beyond the available dimensions of an element are displayed *without* affecting the flow or margins of adjacent elements. Consequently, content of one element may appear to *collide* with the content of its neighbors. Techniques for avoiding this outcome and special cases



caused by rendering issues in IE are discussed in later articles.

`hidden`

Any content which lies beyond the bounds of an element will be hidden from view.

`auto`

The dimensions of an element will be constrained just as when the `hidden` value is used, except that scrollbars will be created as needed to make overflowing content accessible to the visitor.

`scroll`

Both vertical and horizontal scrollbars will be incorporated into the element, even if they're not needed.

The CSS box models: fitting everything together

Now that the fundamental layout properties have been covered, it's time to discover how the width of an element is rendered by the browser according to its CSS properties – and how to keep elements from blowing out your layouts. Some results will make perfect sense, while others will seem horribly counter-intuitive. To complicate matters, there are actually two layout algorithms to consider: the model specified by the World Wide Web Consortium (W3C) in the CSS 2.1 Recommendation, and the one used in older versions of IE.

Choosing the right units for your layout

As in the case of text, elements can be sized with either proportional units such as `%` or `em`, or static units like `px`. Something else to consider is that the browser canvas is always sized at a static value that cannot be assumed without using client-side script code to either retrieve that size, or resize the window – techniques which are ill-suited to the demands of accessibility, usability, and media portability.

The principal rule of sizing elements: mix proportional and static units with care, or not at all

The default value for both `width` is `auto`, which in Standard English is a directive to “use the available space.” The result for block elements is that their computed `width` occupies *all* of that space. With respect to `height`, elements expand to enclose their content by default.

If you change `width` and `height` values, you must then be careful to ensure that the contents of an element will fit (with their margins, borders, and padding) into the width you've specified. The easiest way to do this by engaging in the following process:

1. Consider the largest maximum width likely to be available for your layout, given common display resolutions and/or type sizes. As of this writing, this measurement will typically be something around 800 or 1024 pixels. The broader the expected audience of your site, the more likely it is that the smaller of these values should be chosen.
2. Create a container element for your entire document that is set to an expected width less than the width worked out in step #1.
3. Use the same unit type when setting layout properties for the elements within the container element created in step #2.

Choosing the right unit type for layout: advantages and disadvantages

Table 1: Advantages and disadvantages of the percentage, *em*, and pixel units in specifying layout properties.

Unit	Advantages	Disadvantages
------	------------	---------------

Table 1: Advantages and disadvantages of the percentage, em, and pixel units in specifying layout properties.

Unit	Advantages	Disadvantages
em	<ul style="list-style-type: none"> • Best suited to creating highly precise layout grids in two dimensions • When used in relation to document containers, makes possible layouts that expand or contract according to the size of bodycopy • The computed dimensions of elements become easily predictable 	<ul style="list-style-type: none"> • Fractional units might expand or contract with slight differences between browsers • To achieve the best results, all <code>font-size</code> and <code>line-height</code> values in the document should be set explicit and predictable values
percentage	<ul style="list-style-type: none"> • Best suited to <i>completely</i> flexible layouts • Easiest for creating things like equal columns 	<ul style="list-style-type: none"> • Blowout avoidance might require extra container elements • Might result in unacceptably wide or narrow elements <ul style="list-style-type: none"> • Results are highly dependent on context (see discussion of the box models below)
px	<ul style="list-style-type: none"> • Offers the greatest amount of control over layout • Eliminates most cross-browser variation in layout 	<ul style="list-style-type: none"> • Most ill-suited to accessibility and cross-media support requirements <ul style="list-style-type: none"> • Most susceptible to blowouts

Table 1: Advantages and disadvantages of the percentage, em, and pixel units in specifying layout properties.

The box model components

The box model is really just a series of directives that define how the various layout specifications of an element interact with one another. The components covered by the box model are:

1. document canvas
2. margins
3. borders
4. padding
5. element widths and heights
6. child element properties

The last of these items in turn includes the other five. However, for simplicity's sake this section will focus on simple parent-child element relationships, and reserve discussion of multi-level box model interactions for later articles that will delve into the finer points of page layout.

The W3C box model: everything is additive

The basic rule is that the computed width or height of an element is equal to:

`margin + border + padding + (width|height)`

In many cases the `width` and/or `height` will be set to its default value of `auto`, meaning that the canvas area put aside for content is equal to:

`available_canvas - margin - padding - border`

In such an equation, `available_canvas` is itself a discrete (if often auto-computed) value, less the amount of margins, borders and padding. This number is most important for the *width* of elements, because width calculation errors on the part of a designer will have the undesirable result of causing a horizontal scrollbar to appear in the browser window. Additionally, browsers always place elements at the left margin of the browser canvas that would otherwise overflow beyond the right margin of the browser window, unless instructed to do otherwise.

Consider the following style sheet rule:

```
#myLayoutColumn {
  width: 50em;
```

```

margin: 1.5em auto 1.5em auto;
border: .1em;
padding: .9em;
}

```

As discussed during the explanation of margin properties above, one can expect `#myLayoutColumn` to center itself within its container element, whether that container is `body`, or something created by the production team.

Furthermore, if the activation of “strict mode” (through the use of an appropriate `!DOCTYPE` declaration) causes the W3C box model to be used, one can also expect the computed *non-marginal* width to be:

$$0.1em + .9em + 50em + .9em + .1em = 52em$$

In `screen` media the browser will then take this value, round all of the values separately to the nearest pixel, and render the result accordingly.

Proportional margins and padding in the W3C box model

When the W3C box model is in use, proportional margins and padding are computed relative to the computed width of the *containing* element. To give one example, if you specify `margin: 20%` for an element that’s contained within an element that is 800 pixels wide, the margin rendered around the first element will be 160 pixels on all sides (as 20% of 800 is 160).

If that same element is assigned `padding: 5%`, its computed content width will be 400 pixels:

$$20\% + 5\% + 5\% + 20\% = 50\% \quad 0.50 \times 800 = 400 \quad 800 - 400 = 400$$

Working with document flow

Upcoming tutorials discuss the creation of multi-column layouts, so there are three CSS properties left to introduce in this article: `display`, `float`, and `clear`.

Element types and the display property

With the exception of `html`, `body`, and `table` parts, each element in the HTML 4.01 Recommendation that relates to primary content has an associated type of inline or block. Each type determines default layout behavior in different ways:

inline

- Text and images which immediately follow and/or precede inline elements are rendered on a common baseline with the content of the inline element, depending upon the element(s) in which those elements are themselves placed.
- `margin`, `width`, `height`, and `float` properties in style sheet rules applicable to these elements (except `img` and `object`) are ignored.
- Lines of text within inline elements are laid out with soft linebreaks as needed (or allowed) by the width of their containing block element, except where this behavior is modified by use of the `white-space` property.
- These elements can only contain text or other inline elements.

block

- These elements are rendered as discrete blocks within their containers.
- Unless assigned a `float` value of `left` or `right`, will always be rendered with preceding and following linebreaks.
- Linebreaks between nested block elements that don’t have any content between them will typically be collapsed.

- block elements with a width of `auto` (the default) will always expand to fill the entire width available to them.

The `display` property has three commonly used values – `block`, `inline`, and `none` – of which two refer to the corresponding element types. The effect of changing an element's `display` value is to cause an inline element to behave like a block element, to make a block element behave like an inline one, or to alter the rendering of the document as if the element (and all of its contents) did not exist at all.

As a matter of course, it's vital to know which elements correspond to which types by default, relationships laid out briefly in Table 2:

Frequently used HTML elements and their types.			
Element	Type	Subtype	Notes
a	inline	special	
abbr	inline	phrase	
acronym	inline	phrase	
address	block		Behaves similarly to <code>p</code> in common practice
blockquote	block		Must contain at least one block element when the declared <code>!DOCTYPE</code> is <code>Strict</code>
body			Encloses the entire document canvas; commonly takes on a margin (in IE, Firefox, and Safari) or padding (in Opera) of 10px in screen media
cite	inline	phrase	
div	block		
em	inline	phrase	
fieldset	block		Commonly rendered by default with <code>border: 1px black;</code>
form	block		
h1 ... h6	block	heading	
input	inline	formctrl	
img	inline	special	
label	inline	formctrl	
li	block		Element type not specified in Document Type Definition, but this element may contain either block or inline elements; the complete CSS 2.1 Recommendation sets aside a <code>display</code> value for list items

Frequently used HTML elements and their types.			
Element	Type	Subtype	Notes
<code>ol</code>	block	list	
<code>p</code>	block		May only contain inline elements; commonly rendered with top and bottom margins
<code>span</code>	inline	special	
<code>strong</code>	inline	phrase	
<code>table</code>	block		
<code>ul</code>	block	list	

Table 2: Frequently used HTML elements and their types. Only margins between two adjacent block elements of the same subtype will collapse.

Demonstration 5

How about removing the “Prologue” annotation from the title, just for demonstration’s sake?

Links:

- [Remove the extraneous bit from the title](#)
- [Demo 5 stylesheet](#)

New rules:

```
.sectionNote { display: none; }
```

Causing elements to flow *around* others: the float property



A photo is positioned to the left of this paragraph. Practically all of you will see that the following copy flows naturally *around* it, though some might need first to cease wondering why a well-known science fiction novelist would tape bacon to his cat—even if he was having a slow day. HTML attributes can be used to specify the layout behavior you see, but in this instance the results were accomplished with CSS.

As one can imagine, the property/value pair that works this magic is `float: left;`. The finer points of working with floats will be addressed in later articles, but it’s necessary to touch on the basics here. `float: right` is also a perfectly serviceable property/value pair, and for those occasions when you need to contradict a class assignment that invokes `float`, you can specify `float: none`.

The `float` property *does* come with a few use instructions:

- A `float` value will only matter if it's applied to a block element with an explicit `width`.
- `float`, `clear`, and `margin` properties all appear *together* in style sheet rules meant to create columns within a layout.
- Causing a floated element to stretch to the bottom of its container is a tricky matter, but not impossible. The common way to do this is referred to as [faux-columns](#).

Demonstration 6

Placement of a `float` value on the pullquote has been talked about, so now it gets done and the results can be seen.

Links:

- [Float the pullquote over at the right margin](#)
- [Demo 6 stylesheet](#)

New rules:

```
.pullQuote { float: right; }code
```

Forcing elements *below* their floated predecessors: the `clear` property

Like the `float` property, the `clear` property can be assigned one of the `left`, `right`, or `none` values. The `both` value is also supported.

While the `float` property directs how the content of decedent elements should flow around it, the `clear` property directs how an element should flow around all of its neighbors—in many practical cases, not at all.

Figure 6 illustrates the behavior of `clear: left;` in a layout where two initial consecutive elements have been assigned identical height values, and `float` values of `left` and `right`:



Figure 6: *clear:left* allows the bottom box to clear both columns, as they are the same height.

In the preceding demonstration, the *default* flow of #cLeftWidget would place it just below the greektext – that is, *between* #fLeftWidget and #fRightWidget.

Consider what happens when the first of the same collection of elements is made shorter than its flush-right sibling, as seen in Figure 7.

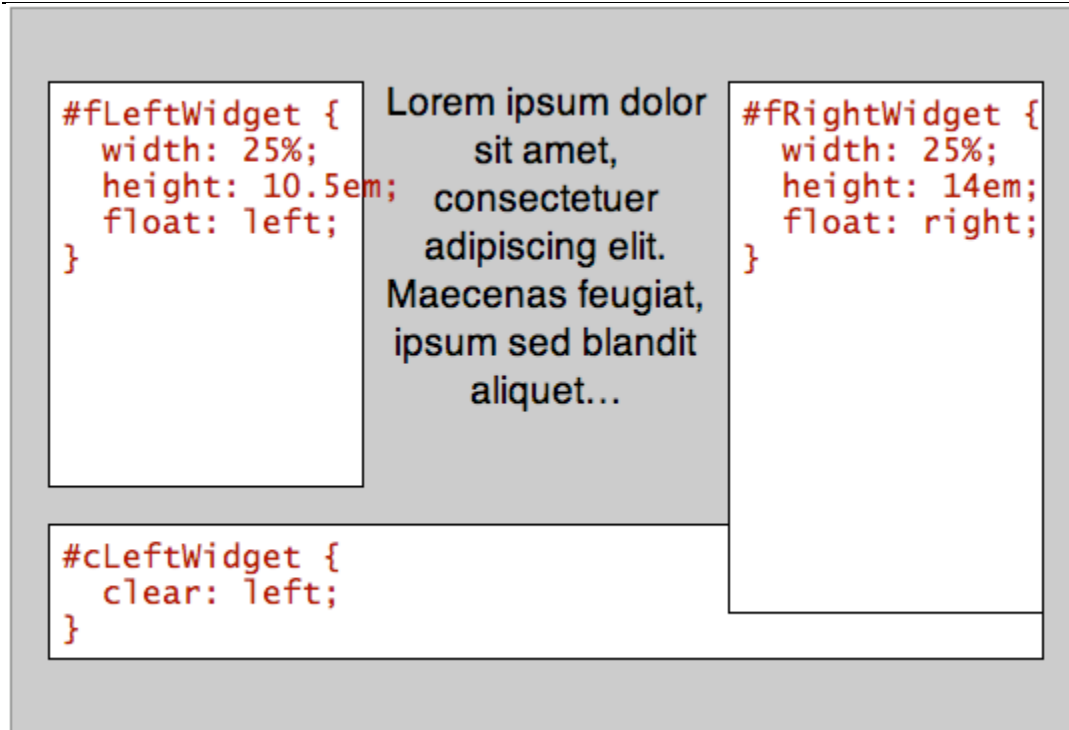


Figure 7: When the right column is longer than the left column, `clear:left` will not clear both columns and so `clear:both` must be used instead

In the first example, the `clear` value of the trailing element is set to `left` in order to make a point: because both of the floated elements are the same height, the cleared element will be pushed below both. However, the second example proves that in order to achieve the same result with floated elements of differing heights, `clear: both;` must be used.

This discussion of the `clear` property is intended as a simple introduction to its effects, while later articles discuss the finer points of technique associated with its use.

Summary

Between differences in rendering engines, the need to cover a wide swathe of traditionally defined ground, and the inability to predict the certain dimensions of a browser window, the layout of Web documents is fraught with hassles and caveats. However, the common level of CSS support has advanced to the point where Web documents are not hard to get to give decent results across browsers.

Further reading

- Bergevin, Holly, and Gallant, John. 2006. Explorer exposed. Position Is Everything. <http://positioniseverything.net/explorer.html> (accessed 1 July 2008).
- Bos, Bert, *et al.* 2007. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. World Wide Web Consortium. <http://www.w3.org/TR/2007/CR-CSS21-20070719> *etc.* (accessed 30 June 2008).
- Raggett, Dave, *et al.* 1999. HTML 4.01 specification. World Wide Web Consortium. <http://www.w3.org/TR/1999/REC-html401-19991224> *etc.* (accessed 30 June 2008).
- Raymond, Eric, and Steele, Guy, eds. 2003. Brute force. The Jargon File (Version 4.4.7). <http://www.catb.org/jargon/html/B/brute-force.html> (accessed 30 June 2008).
- Scalzi, John. 2006. Clearly you people thought I was kidding. Whatever. <http://www.scalzi.com/whatever/004457.html> (accessed 30 June 2008).

Exercise questions

1. Under which circumstances is it best to use the shorthand `margin` value, or a single margin property such as `margin-top`?
2. When the shorthand `margin`, `padding`, and `border-width` properties are provided with all four values, in what order are those values applied to the four sides of an element?
3. If you want to place a rule under the text of each heading in a document, which property would you use?
4. Which `border-style` value would you use to give an element an appearance like an interface button?
5. *Yes or no:* Will specifying a border around an element will also provide for a gutter around the content of that element, by default?
6. If you create an element that isn't as wide as its container, which property/value pair do you need to set to ensure that the element is horizontally centered within its container?
7. *Yes or no:* If you place a container element within `body` and set its `width` to a value greater than 100%, will the behavior of the document canvas change?
8. If an image is too large for its containing element, which property/value pair would you use to ensure that your page layout doesn't blow out, and why?
9. If you assign a `display` value of `block` to an a (link) element and give that element a reasonable height and width, how does the mouseover behavior of that link change in `screen` display media?
10. Under normal circumstances, a block element expands to fill the width of its container (less margins, borders, and padding). By default, does this behavior truly change when that element is preceded by a floated element – or merely *appear* to change?
11. If you intend to apply a `float` value to an element, which other property must you also set on that element?
12. If you wanted to make *absolutely* sure that an element would *always* expand to fill the width of its container, which property/value pairs would you set?

About the author



Ben Henick has been building Web sites in one capacity or another since September 1995, when he took on his first Web project as an academic volunteer. Since then, most of his work has been done on a freelance basis.

Ben is a generalist; his skillset touches on nearly every aspect of site design and development, from CSS and HTML, to design and copywriting, to PHP/MySQL and JavaScript/Ajax.

He lives in Lawrence, Kansas, with three computers and zero television sets.

You can read more about him and his work at henick.net.

31: CSS background images

BY [NICOLE SULLIVAN](#) · 26 SEP, 2008

Introduction

Admit it! Since the first article in this course, you've been itching to learn how to make your site look fierce and fabulous. Maybe you even skipped ahead to this section?

Background images are all about making your site look sexy, but you might be surprised how closely they build upon the fundamental concepts you have already learned.

As you already learned earlier on in the course, one of the most important changes that comes with CSS was the ability to separate *presentation*, or the way things look, from *semantics*, or what things mean. The CSS background image is among the most important tools you have at your disposal, because it lets you apply decorative images to particular parts of your HTML without adding any extra weight to your HTML. Previously, authors (that's you!) were forced to fill their code with `img` tags.

CSS and in particular the `background` property keep your HTML free from presentational clutter. Redesigns and other transitions, in the life of sites built with modern methods, can then be completed much more smoothly. You'll be able to update your entire site by changing only the style sheet, rather than recoding every HTML page. Depending on the size of your site, this can be a substantial saving.

In this article I'll show the basics of how CSS background images work, including applying a background image via CSS, adjusting its placement, tiling it vertically or horizontally and combining background images using [CSS Sprites](#) to improve [site performance](#).

The article structure is as follows:

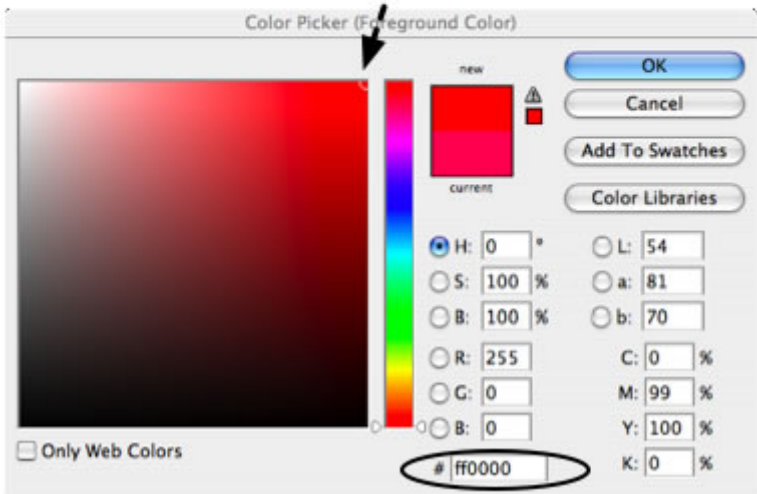
- [How does it work?](#)
 - [Background properties](#)
- [Building an alert message](#)
 - [The code](#)
 - [Creating the CSS hook, or selector](#)
 - [Adding the background color](#)
 - [Apply the background image](#)
 - [Controlling background repeat](#)
 - [Attachment](#)
 - [Positioning the image](#)
 - [Using shorthand to pull the whole thing together like a pro](#)
 - [Experimenting with code](#)
 - [testing for quality](#)
- [Sprites](#)
 - [A complex sprite and background image example](#)
 - [Creating the sprite](#)
- [Summary](#)
 - [Image credits](#)
- [Exercise questions](#)
- [Further reading](#)

How does it work?

The CSS for backgrounds is split into several different properties. Using these properties, such as `position` and `color`, you can begin to control the look and feel of your page. In this article, you will go through CSS background images in detail, building an alert message as an example, step by step.

First, let's learn a little more about the different properties at our disposal.

Background Properties

Property	Definition	Description
background-color	Sets the background color of the HTML element.	<p>There are several ways to indicate <code>background-color</code>, including RGB values and keywords. Most people use hexadecimal notation, a pound/hash symbol (#) followed by six characters. The first pair indicates the red levels, and the second and third indicate the green and blue levels respectively—<code>#RRGGBB</code>.</p> <p>Many color picker tools will help you find the hexadecimal notation of a given color. Pure red, for example would be <code>#FF0000</code>.</p> <p>Tools like Photoshop will allow you to choose a color, such as red, and will give you the hex value.</p>  <p>Valid values include a color value, <code>transparent</code>, or <code>inherit</code>.</p>
image	Indicates the path <i>or URL</i> of the background image.	<p>Set the <code>background-image</code> by showing the browser where to find the image, using the URL. For example; <code>url(alert.png)</code>. Note that the path is prefaced with the keyword <code>url</code> and wrapped in parenthesis. This syntax is important to the browser understanding that you mean to indicate a location.</p> <p>Valid values include a URL, <code>none</code>, or <code>inherit</code>.</p>
repeat	Indicates in which direction the background image should be tiled.	<p>Images can be tiled vertically, horizontally, or both, to fill the entire width or height of an HTML element. Use <code>background-repeat</code> to instruct the browser to repeat a background image.</p> <p>Valid values include <code>repeat</code>, <code>repeat-x</code>, <code>repeat-y</code>, and <code>no-repeat</code>.</p>
attachment	Sets the behavior of the background image when the user	<p>Images can either scroll with their content, or stay fixed in place in the view screen. Valid values include <code>scroll</code>, <code>fixed</code>, and <code>inherit</code>.</p>

	scrolls.	
position	Tells the browser where to position the background image.	<p>Images can be displayed anywhere within the borders of the HTML element on which they are applied. Use <code>background-position</code> to precisely place your images for visual effect and layering.</p> <p>There are many useful ways to indicate background position, keywords and numeric values. Keywords (such as <code>top</code> and <code>bottom</code>) are very useful and easy to read. Pixel values are very precise, but don't adapt to changing heights and widths. Negative pixel values are useful when using CSS sprites, as you'll find out later.</p> <p>When percentages and pixels are used, the starting point is always the top left corner of the HTML element, although the way image positioning works with pixels and percentages is rather different. Pixels always move the image a set number of pixels away from the top and left of the containing box (or towards if they are negative values), regardless of the size of the image and the containing box. Percentages on the other hand move the image a percentage of the difference between the containing box size, and the image size. If the image and the containing box are the same size, percentages won't move the image at all.</p> <p>Valid values include <code>length</code> (generally in pixels), <code>percentage</code> (of the width of the element), and the keywords <code>top</code>, <code>right</code>, <code>bottom</code>, <code>left</code>, and <code>center</code>. Note that <code>center</code> can be used to indicate both vertical and horizontal center. Note also that you can mix percentages and pixels in rules, but not keywords and pixels or keywords and percentages.</p>
background	The shorthand property that can be used to describe all the other properties in one line.	<p>Shorthand properties are very nifty indeed. Most developers use them to keep the CSS as lean as possible and group related properties. You can write a general rule using shorthand, and then override it as needed with specific properties.</p> <p>The properties should always be indicated in the same order, to allow browsers to easily interpret the intended styles:</p> <ol style="list-style-type: none"> 1. <code>color</code> 2. <code>url</code> 3. <code>repeat</code> 4. <code>attachment</code> (very rarely used; may be omitted) 5. <code>horizontal-position</code> 6. <code>vertical-position</code> <p>An example of this shorthand with all the properties used (except <code>attachment</code>) is as follows:</p> <pre>background: green url(logo.gif) no-repeat left top;</pre>

Building an Alert message

Now I've gone through the basic syntax involved, I'll walk you through building up a complete alert box example, which will demonstrate all the aspects of background images.

The design

Let's say a graphic designer has provided a visual mock-up of the alert message you want to create for your web site. Looking at the alert you see that the background is light orange, setting it off from the surrounding paragraphs. It also has an alert icon ten pixels from the top left corner.

Note that the mockup has one line of text, but it might contain more in the future. One of the most important skills of the web developer is to anticipate how a design will evolve. Part of respecting the artistic vision for a site is thinking about consistency from launch to redesign. So the alert message could contain more than one line of text, or even multiple paragraphs, lists, or other HTML elements. You should try to be as element agnostic as possible. This will increase the likelihood of code reuse, and set up the site to be as fast and efficient as it can be. The mockup looks like Figure 1.

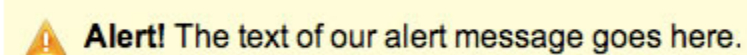


Figure 1: The graphic designer's mockup of our alert box.

The designer has also provided the icon we are meant to use, as shown in Figure 2.



Figure 2: The alert icon.

The code

Based on what you have learned about CSS backgrounds in the first part of this article, you may already be thinking about how to build this alert message. I'd like to encourage you to have a go at it now, and then compare your work against my example.

Ok—had a try? Now let's go through it step by step. Each screen shot links to code examples, so you can check out the source at each stage. Experiment with the code, increase or decrease values, and try out alternatives. You may also want to follow along, writing each new line of code in a tool such as Opera Dragonfly or Firebug, so you can immediately see the results of each step.

Creating the CSS hook, or selector.

First you need to create a class `alert`, for the CSS to hook on to. Create new CSS and HTML skeleton files, link the CSS to the HTML file, and add the following code to them:

The CSS:

```
.alert { ... }
```

The HTML:

```
<p class="alert">
  <strong>Alert!</strong> The text of our alert message goes here.
</p>
```

Here I am styling the alert with a `class`, rather than an `id`, because I could have *more than one alert* in the page, for example a form element with several errors. You want to make your CSS as flexible as possible and constrain things to correspond to the design when building the HTML.

Ok, so you've put a solid foundation in place, but it still looks like an ordinary paragraph because you haven't yet added any styles. Lets do that next.

Note: I have intentionally chosen *not* to limit the class `alert` to paragraphs; alert boxes could easily contain other elements as well. You should leave as much flexibility as you can within your CSS.

Adding the background color

You already learned about using [background color in text treatments](#) in Article 29, *Text styling with CSS*. The same principles apply to any HTML element and can be combined with background images to create visual effects. If the background color has neither been set nor inherited, it is, by default, transparent.

Let's add the light orange background color to the alert box to make it stand out from the text around it. You don't want it to be too dark because it is important that you keep a reasonable level of [contrast between the text and the background color](#). Add the following property inside your CSS rule

```
.alert{background-color: #FFFFCC;}
```

The Alert box should now look more like Figure 3.

Alert! The text of our alert goes here. Adding the background color really helps our alert stand out. You may have noticed that I've also added a small amount of space between the sides of the alert box and the text.

Figure 3: An alert box with background colour added.

Applying the background image

Now let's add the image to the alert. The path to the background image needs to be wrapped in `url()`, as shown in the code below. Add the highlighted line to the CSS rule.

```
.alert{  
  background-color: #FFFFCC;  
  background-image: url(alert.png);  
}
```

The alert box will now look like Figure 4.



Figure 4: The background image has been added, but the tiling looks awful.



Remember that each background property has a default value—if you haven't specified a value, the default will be applied. Of course, you will have noticed that the image is tiling over our entire alert, much like mosaic tiles on a kitchen floor. What is the takeaway? Background images are set to repeat both horizontally and vertically by default. Repeating backgrounds are particularly useful for gradients and patterns that fill the screen or a particular HTML element, but that effect is *not* desired in this case.

Controlling background repeat

Figure 5: Much like our background image, these tiles repeat both horizontally and vertically.

Reading specifications can certainly be intimidating, but the specification is a really good place to figure out how CSS is *supposed to work* before you delve into the myriad browser differences. Go take a look at the [colors and backgrounds portion of the W3C Specification](#) and try to find the keyword to use when you don't want a background image to be repeated. We'll use that in our example below.

Found it? Note that there is a section for each background property including `background-repeat`. Under Value, you'll see all the possible choices including; `repeat`, `repeat-x`, `repeat-y`, `no-repeat`, and

`inherit`. By default (initial) background images are set to repeat. No direction is specified, which means that the image will tile both horizontally and vertically. You have most likely guessed that `no-repeat` is the value you are looking for to prevent the image from tiling in either direction. Add the following highlighted line to the CSS rule.

```
.alert{  
  background-color: #FFFFCC;  
  background-image: url(alert.png);  
  background-repeat: no-repeat;  
}
```

The alert box will now look like Figure 6.



Alert! The text of our alert goes here. Phew! That's getting better.

Figure 6: The alert box, with a single copy of the background image (no tiling.)

You don't need to know the size of the HTML element; you simply cut a slice from your gradient and set it to repeat in the direction you want; either `x` for horizontal, or `y` for vertical. Additionally, you can choose to repeat in both directions (like kitchen tile) or neither direction. Gradients often repeat horizontally or vertically (see Figure 7,) patterns often repeat in both directions, and icons usually do not repeat. You will explore `background-repeat` further in a later example.



Figure 7: The greenish yellow tiles in this example repeat only horizontally.

Let's take a look at a practical example from my website—look at Figure 8.

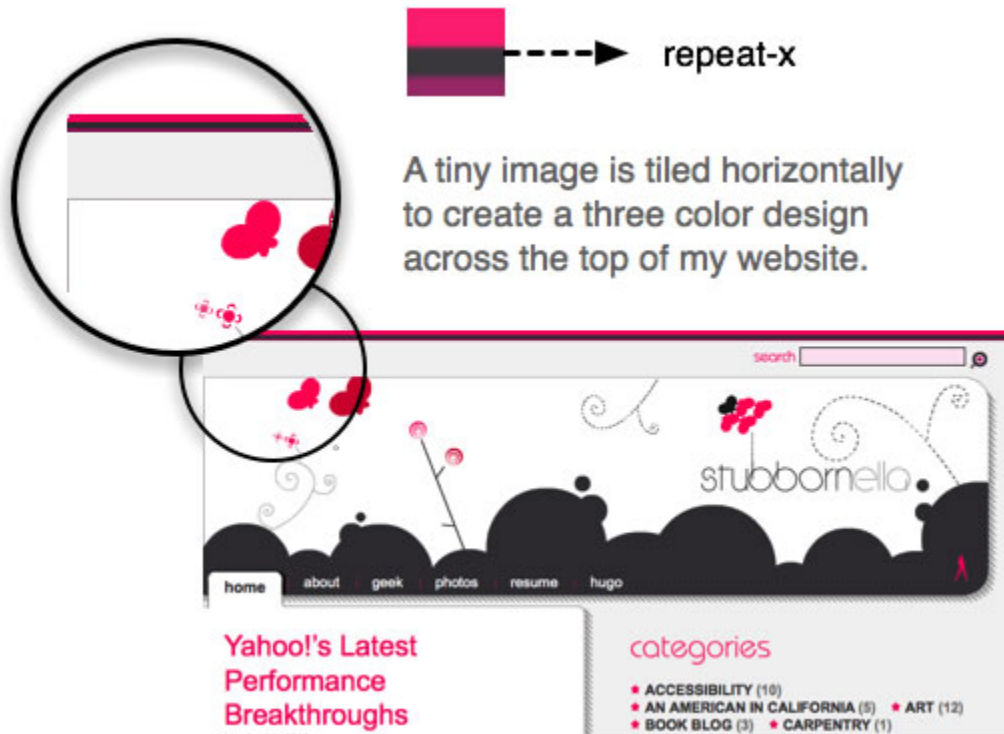


Figure 8: A repeating example from my own web site.

The CSS I used to add this decorative effect is relatively simple. I made the background repeat horizontally using `repeat-x`:

```
body{background-repeat: repeat-x}
```

Attachment

`attachment` allows you to specify how the background behaves when the user scrolls down the page. The default behaviour is `scroll`, which causes the background image to scroll along with the content.

On the other hand, setting `background-attachment` to `fixed` causes the element to be stuck to the browser window, so it stays in the same place when the content inside the element it is attached to is scrolled. This creates some odd effects as the image will only be visible when the HTML element it is attached to is scrolled over it. The W3C uses it to mark the status of their specifications, for example the ["W3C Candidate Recommendation" image at top left](#). Scroll down the page and the image stays top left. It is attached to the `body` element, so it is always visible.

This step will have no effect on our display, because browsers set background images to scroll by default, but let's add it to the code anyway so that you can see how the property is used. Add the highlighted line to the CSS rule:

```
.alert{  
  background-color: #FFFFCC;  
  background-image: url(alert.png);  
  background-repeat: no-repeat;  
  background-attachment: scroll;  
}
```

As shown in Figure 9, the visual display of the alert box is not much different to how it was before.



Alert! The text of our alert goes here. Our alert box is looking pretty good now, but it is still a little strange because the tiny background image is stuck to the upper left hand corner of our box.

Figure 9: Not much different here.

Positioning the image

Positioning is the fine tuning that lets you place your background image exactly where you want it to be, both horizontally and vertically, within the HTML element. This property takes keyword and number values such as `top`, `center`, `right`, `100%`, `-10%`, `50px` and `-30em`.

Figure 10 shows the values you might use to place your background images in different positions.

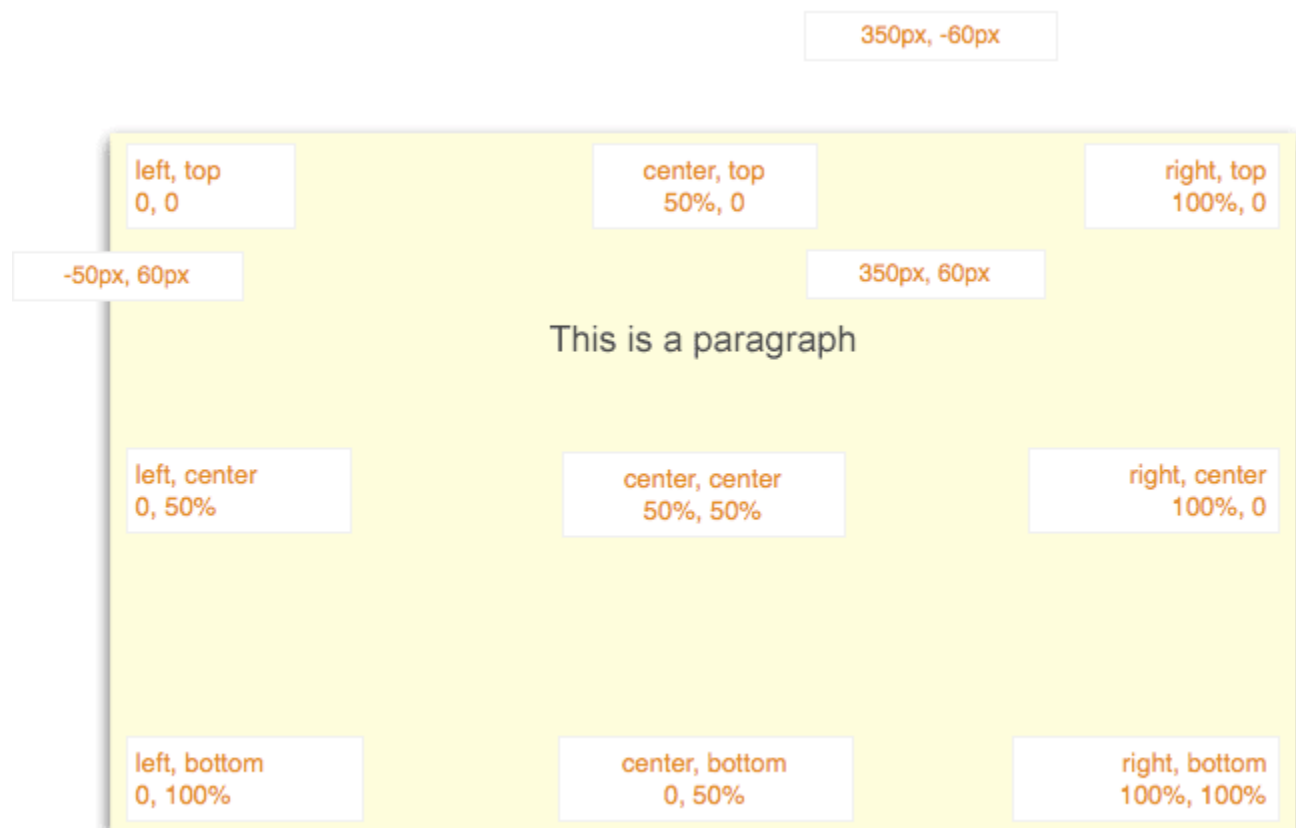


Figure 10: Various examples of background position using keywords, percentages, and pixels.

So let's position the background image. You want it to be in the top left corner, but not touching the sides, so you need to offset it by 10 pixels from both the top and left—this can be done by adding the following highlighted line to the CSS rule. Do this now.

```
.alert{
  background-color: #FFFFCC;
  background-image: url(alert.png);
  background-repeat: no-repeat;
  background-attachment: scroll;
```

```
background-position: 10px 10px;
}
```

The first value is the horizontal offset, the second is the vertical. In this case they are the same. Your alert box should now look like Figure 11.



Alert! The text of our alert goes here. Note that we include the word "alert" too so that we have a text equivalent for all this visual goodness.

Figure 11: Using positioning to place the background image.

Tip: Stick to either keywords or number values—older browsers may ignore your declaration if you use both at once. Using `right` and `bottom` will achieve the same thing as `100%` horizontally or vertically, respectively.

Using shorthand to pull the whole thing together like a pro

As you have already seen, certain CSS properties can be grouped together. Background and all of its sub properties are among them. The CSS code we've written so far can be rewritten in shortened form, as follows:

```
.alert{background: #FFFFCC url(alert.png) no-repeat scroll 10px 10px;}
```

Tip: When grouping sub properties of `background`, always put the properties in the following order—this is important for both cross browser compatibility and stylesheet organization and maintenance:

1. color
2. image
3. repeat
4. attachment
5. horizontal position
6. vertical position

Try replacing the old CSS with the shorthand shown above, and your example should look exactly the same—see Figure 12.



Alert! The text of our alert message goes here. Our final alert box is looking pretty slick, the CSS code is lean and efficient.

Figure 12: The shorthand works like a charm!

Experimenting with the code

The best way to remember all the nuances of CSS is to try out the options yourself—try changing some of the properties in the example, and see how that affects it. Set the `background-position` to `100% 100%`, and notice that it gives the same result as using the `right` and `bottom` keywords. What about if you change it to `-5px 0`? Why do you think you now can't see part of the image?

Testing for quality

Testing is extremely important to providing a good user experience. Just because the site looks good on your machine with your specific configuration doesn't mean that it will look good for everyone. You should follow these basic minimum steps when testing your alert box.

- Increase or decrease the amount of text inside the alert.
- Increase the text size in your browser at least two levels. Would it have been better to use ems to position our image? Then what happens when you increase the text size?
- Apply the `class` `alert` to other elements such as `div`, `p`, `ul`, `strong`, or `em`. What do you need to change to make the class agnostic?
- Include several paragraphs and a list inside an alert `div`—does the code still work?
- Verify the alert visually in the [Grade 1 browsers](#) (also known as A-grade). My advice is to write for good browsers and adapt for Internet Explorer once the code works.

Rigorous testing is part of learning to write CSS. The more careful you are while learning, the faster you will become.

Sprites

Users want it all. They want your site to be glamorous, interactive, and also fast, however including large numbers of CSS background images can slow your site down considerably—the more HTTP requests you make, the slower your site will be (an HTTP request is when your computer is accessing a web site and needs to ask the server to send it another asset that makes up the site, such as a CSS file or image - each additional request means a longer loading time for the site). To get around this limitation, you can combine related icons into a single image, known as [CSS Sprites](#). The `background-position` property allows you to then place the image in the appropriate positions so the icons display through the *window* of the HTML element the CSS sprites are attached to.

For example in Figure 13, you will see that to display the earth icon through the HTML window you can place the image using `left top`. To move the position of the image so the alert icon is displayed, the background position needs to be changed to `-80px 0`. The negative horizontal value pulls the image to the left.

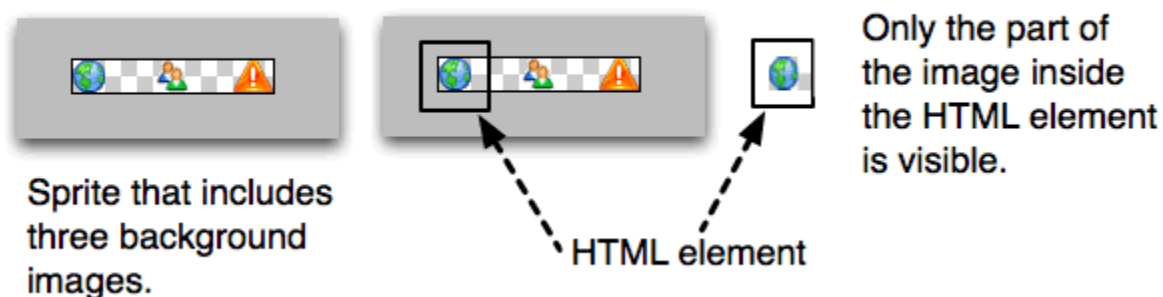


Figure 13: Using CSS Sprites to reduce HTTP requests.

Note: If you use negative background positions, Safari will repeat your image, even if you've specified `no-repeat`. This is something to keep in mind as you start playing with background images to create more complicated layouts.

A complex sprite and background image example

Let's have a look at how CSS sprites can be used to good effect. Suppose our friendly designer sent us a new mockup. This one is for a list of links on the landing page of a blog. It points to the bloggers' LinkedIn profile, RSS feed, Flickr photos, and bookmarks. Looking at each link, we realize that there is a gradient starting in the center as white and going to gray at the top and bottom of the link, and to further

complicate things the designer asked if we could make each link plain white with no curve when visitors hover over the link—check out Figure 14.

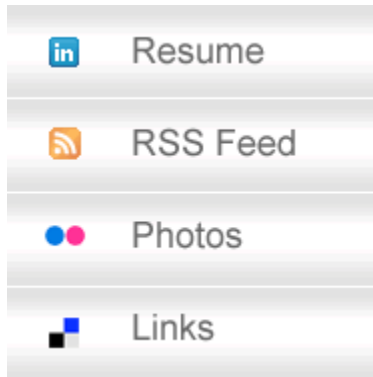


Figure 14: The new design mockup.

The logos could be included using `img` elements in the markup, however using CSS sprites is a much better way to go—the sprites load faster as only one image needs to be loaded (not four), and it declutters your HTML, reducing the amount of markup needed.

Creating the Sprite

The first step is to cut out the four logos and create the sprite set, as seen in Figure 15.



Figure 15: The sprite set.

You also need to cut out a 1 pixel wide slice of our gradient. For the sake of visibility, I have cut out a slightly larger slice, but you only need one pixel—see Figure 16.



Figure 16: The slice for our gradient background.

The HTML for the list is an unordered list containing links. Note the empty `span` elements inside the links. It is very important not to have fixed `height` and `width` on elements that contain text—after all, you have no idea how large the text will be. What happens if the site gets translated to German? You can use these extra spans to display the logos. As an alternative, you may decide that you don't want to have extraneous non-semantic markup cluttering up your HTML. In this case you will need to use a larger sprite and leave white space between the icons. Keep in mind that this will be slower for users on slow connections, especially those on mobile phones. The code for the list looks like so—add this to an HTML template:

```
<ul class="navigation">
  <li id="resume">
    <a href="#"><span></span>Resume</a>
  </li>
  <li id="rss">
    <a href="#"><span></span>RSS Feed</a>
  </li>
  <li id="photos">
    <a href="#"><span></span>Photos</a>
  </li>

  <li id="links">
```



```
    <a href="#"><span></span>Links</a>
  </li>
</ul>
```

The CSS makes use of both background images. First, take a look at the gradient background image. There are three interesting things to note about it:

1. The first is that the image repeats horizontally (`repeat-x`.) This is how we are able to make such a small image spread across the entire list.
2. The second is that the image is centered vertically. You want the round bit of the image to appear in the middle of the list item, so you should use a background position of `left center`.
3. Finally, in the CSS I've applied a background color that is the same grey as the grey in our gradient image. In this way, if the element grows, it won't look broken. For more information about this kind of technique I recommend [Bulletproof Web Design](#) by Dan Cederholm.

Add the following CSS to a new CSS file, and link it to the HTML file:

```
.navigation, .navigation li {
  margin:0;
  padding:0;
}

.navigation li {
  border-top: 1px solid white;
  list-style-type:none
}

.navigation li a {
  background: #E2E2E2 url(sprite gradient bkg.png) repeat-x left center;
  padding:20px;
  display:block
  font-family: Arial, Helvetica, sans-serif;
  color:#333;
  font-size:18px;
  text-decoration:none
}

/* hover effects */

.navigation li a:hover, .navigation li a:focus{
  background: transparent none;
}
```

The last line means that the element should have no background color or image when the user hovers using the mouse, or focuses using the keyboard. Perhaps you are wondering why I applied the background properties to the link rather than the list item? The answer is that Internet Explorer 6 and earlier do not support pseudo classes like `hover` on elements other than links. I've made the adjustment to accommodate this constraint.

Next you can create the CSS for the little logos. As usual, you can start by defining the most general case for all `span` elements within your navigation module. It is here that you define the image to be used by all spans, the repeat, and the background position (each is different, so let's use the first). You can use shorthand for this rule. Note that I'm using CSS comments to divide sections of our code into manageable chunks. Add the following code to the bottom of the CSS file:

```
/* general case */

.navigation span {
  background:url(sprite logo.png) no-repeat left top;
  height:15px;
  width: 15px;
  margin-right:20px;
```



```
display:-moz-inline-box;  
display:inlineblock;  
}
```

With the general case well in hand, you can now define the exceptions, or *what is different* about each specific logo. In this case, the only CSS that changes is the `background-position`. Each respective list item needs to have the image pulled 15 pixels more to the left, because each of the logos are 15 pixels wide. Add the following to the bottom of the CSS file:

```
/* exceptions */  
  
#rss span {  
    background-position: -15px 0;  
}  
  
#photos span {  
    background-position: -30px 0;  
}  
  
#links span {  
    background-position: -45px 0;  
}
```

This example might seem intimidating at first. Keep your focus on the background images. In this case, I've used negative pixel values to pull the background image left so that the relevant part of the image is seen. Positive values push the background image down and right, negative values pull the image up and left.

Play with the background position values in the [finished example](#), to better understand how to adjust sprite positioning.

Summary

You should now understand CSS background images, and what's more, you are becoming more comfortable reading specifications, so if you have doubts about a particular property, you should know how to go look it up. This article covered background color, image, repeat, attachment, and position. You also learned why developers use CSS Sprites, and how to use this advanced technique.

Image credits

- [The Tiles](#), by DimsumDarren
- [little glass tiles](#), by emdot

Exercise Questions

- A paragraph is 40px by 180px and your background image is 60px by 200px. Will you see the entire image or only part of it? Why?
- You want an image to be positioned in the bottom left corner of the `blockquote` element—please fill in the correct values.

```
blockquote{background: yellow url(quote.png) no-repeat scroll ____ ____ ;}
```

- Say you wanted each `h2` in your document with a `class` of "question" to have a gradient pattern applied. Would you use `repeat-x`, `repeat-y`, `no-repeat`, or `repeat` to achieve something similar to the example below? Why?

Example Heading Level 2

- What would be the background position of the example in question number 3? How could you creatively use a background color to be sure the background could expand to any height? Why is this important?
- What shorthand can you use to remove all background properties?
- What is the purpose of CSS sprites?

Further reading

- [Performance and HTTP requests on YDN](#)
- [CSS2 Specification—Colors and Backgrounds](#)
- [CSS Sprites—Image Slicing’s Kiss of Death](#)
- [Bulletproof Web Design](#)—My favorite book

About the author



Nicole Sullivan is a CSS performance guru living in California. She began her professional career in 2000, when her future husband (then a W3C employee) told her that if her website didn’t validate he wouldn’t be able to sleep at night. She thought she’d better figure out what this “validator” thing was all about, and a love for standards was born.

As her appreciation for performance and large-scale sites grew, she went on to work in the online marketing business, building CSS framework solutions for many well known European and worldwide brands such as SFR, Club Med, SNCF, La Poste, FNAC, Accor Hotels, and Renault.

Nicole now works for Yahoo! in the Exceptional Performance group. Her role involves researching and evangelizing performance best practices and building tools like YSlow that help other F2E’s create better sites. She writes about standards, her dog, and her obsession with object oriented CSS at www.stubbornella.org.

32: Styling lists and links

BY [BEN BUCHANAN](#) · 26 SEP, 2008

Introduction

Many elements on a webpage are a little bit "forgiving" in terms of design—if they're not "just right" it doesn't matter too much. With lists and links it's a different story—if you don't get them right, you can cause some serious problems for people trying to use your website.

Links in particular have some key style requirements and user expectations. Poorly styled links can ruin the user experience on a website, as people have to stop and think about where to click. In the worst case, a user might not even be able to tell which items on the page are actually links.

In this article we'll look at the core skills you need to create robust list and link styles. We'll also discuss some ways to avoid key pitfalls of these elements and produce a final result that will work across different browsers, and be accessible to users with disabilities.

There are a number of examples used in this article, so you can [download the lists and links example files](#) to follow along with them.

The article contents are as follows:

- [Styling lists](#)
 - [Basic bullets and numbers](#)
 - [Custom bullets using images](#)
 - [List margins and padding](#)
 - [Unordered lists](#)
 - [Ordered lists](#)
 - [So, what to do?](#)
 - [Using list-style-position](#)
 - [What about definition lists?](#)
 - [Nested lists](#)
 - [Horizontal lists](#)
 - [Faux columns](#)
 - [Legacy browsers](#)
 - [Lists conclusion](#)
- [Styling links](#)
 - [Understanding link states](#)
 - [How browser evolution set expectations](#)
 - [User expectations](#)
 - [Use color carefully](#)
 - [Getting down to business: the CSS](#)
 - [Styling link states in the right order](#)
 - [Styling lists](#)
 - [Controlling defaults](#)
 - [Underlining](#)
 - [Outline](#)
 - [Example: recreating the Netscape defaults](#)
 - [Faux underlines using border-bottom](#)
 - [Styles that don't rely on color](#)
 - [Icons on links](#)
- [Bringing it all together—a simple navigation menu](#)
- [Summary](#)
- [Exercise questions](#)
- [Further reading](#)

Styling Lists

First, I'll take you through the basics of styling lists with CSS, before then moving on to look at some slightly more complicated techniques.

Basic bullets and numbers

The fundamental thing to consider when creating a list style is what form of bullet or numbering you want to use. You might also choose to remove the bullets and numbers completely. As you learned in the [HTML Lists article](#), there are many options available, set using the `list-style-type` property.

For example, to set all unordered lists on your site to use square bullets, use this CSS:

```
ul li {  
  list-style-type: square;  
}
```

Which will produce something like Figure 1:

- List item
- List item
- List item

Figure 1: Unordered list with square bullets.

Some common list types are shown in Figure 2:

Unordered lists

Disc

- First item
- Second item
- Third item

Square

- First item
- Second item
- Third item

Circle

- First item
- Second item
- Third item

None - no bullets

- First item
- Second item
- Third item

Ordered lists

Decimal

1. First item
2. Second item
3. Third item

Decimal with leading zeros

01. First item
02. Second item
03. Third item

Lowercase ascii letters

- a. First item
- b. Second item
- c. Third item

Lowercase roman numerals

- i. First item
- ii. Second item
- iii. Third item

Figure 2: Common list styles.

The [basic list choices example page](#) shows some more options.

Note that the bullets and numbers will be rendered using the `color` which is set for or inherited by the `li`. If you need the bullet to be a different colour from the text, you will need to use an image instead, or work around the issue using other elements within the list items (this may be easy if all the items are links, for example).

Custom bullets using images

The standard set of bullets are enough for basic content, however it is a common design request to replace them with a custom image.

The CSS specification does include the `list-style-image` property for adding a custom list image. However, the property has limited positioning options for the background image, and in some circumstances doesn't work at all in IE. So it has become a far more common practice to simply set a background image on the list items.

Let's imagine you have a list of RSS feeds and you want to change the bullet to the standard orange RSS icon. We'll give the list the class "rss" to differentiate it from other lists:

```
<ul class="rss">
  <li><a href="http://example.com/rss.xml">News</a></li>
  <li><a href="http://example.com/rss.xml">Sport</a></li>
  <li><a href="http://example.com/rss.xml">Weather</a></li>
  <li><a href="http://example.com/rss.xml">Business</a></li>
  <li><a href="http://example.com/rss.xml">Entertainment</a></li>
  <li><a href="http://example.com/rss.xml">Funny News</a></li>
</ul>
```

First we'll set the list to have no `list-style-type` and remove the margin and padding. Then, simply add a background image to each list item, some left padding to move the text over to let the image show through, and some bottom padding to space out the list items:

```
.rss {
  margin: 0;
  padding: 0;
  list-style-type: none;
}

.rss li {
  background: #fff url("icon-rssfeed.gif") 0 3px no-repeat;
  padding: 0 0 5px 15px;
}
```

This will produce a list with the RSS image instead of bullets, as seen in Figure 3:

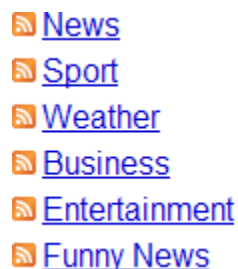


Figure 3: The list with image bullets.

Note that the background image is positioned using pixels for a precise placement. Depending on the design you are creating, you might also be able to use %, ems or keywords. Just be careful when your design features content that might cause a list item to wrap over several lines—if you set the background to vertical center or 50% it can look quite strange, as shown in Figure 4:

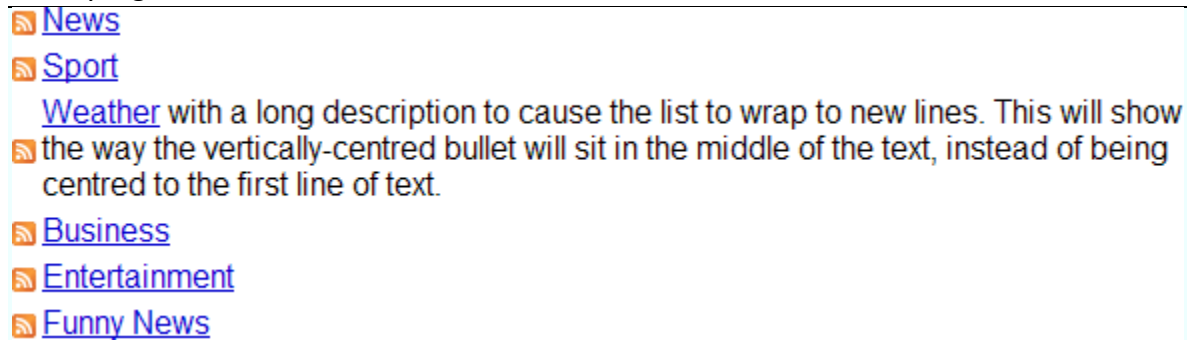


Figure 4: A demonstration of vertically-centred bullet images on a multi-line list item.

By setting the image to sit at the top of the list item, you maintain the default bullet behaviour (where the bullet sits on the first line)—see Figure 5:

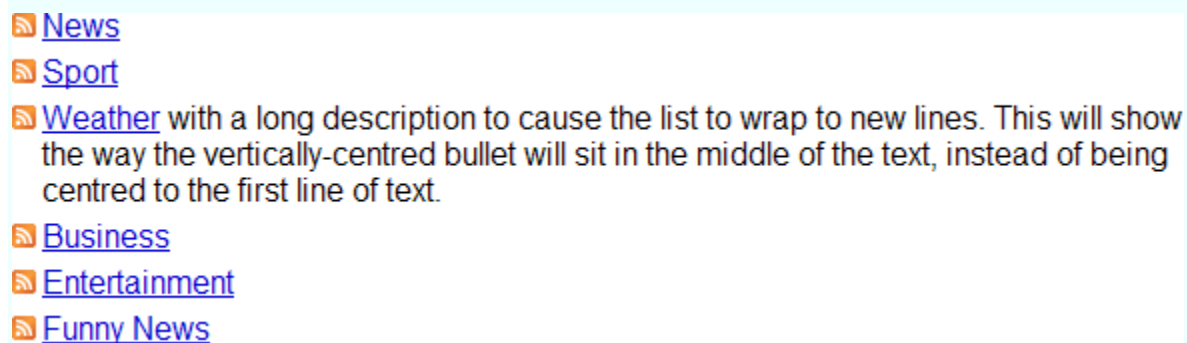


Figure 5: A demonstration of top-aligned bullet images on a multi-line list item.

List margins and padding

Clever use of margins and padding can make lists look much more polished and professional, but you need to know what you are doing, and also bear in mind that the situation differs between different types of list. In this section I'll take you through applying sensible margins and padding to the two most common types of list.

Unordered lists

One thing you will probably notice quite quickly is that the default style for lists indents them more than the default style for paragraphs—see Figure 6:

A paragraph for reference.

- First item
- Second item
- Third item

A paragraph for reference.

Figure 6: Default styled lists are indented on the left.

If you want your unordered list items to sit at the same left-align point as the rest of your content, you will need to set some styles to control the indentation to your liking. Different browsers require different settings—some need the margin removed, others need the padding removed. So, to reset for all browsers, reset both:

```
ul {
```

```
margin: 0;  
padding: 0;  
}
```

This may not have the effect you were expecting, as it will get the *text* to sit flush left but the bullets will sit outside the text, as shown in Figure 7:

A paragraph for reference.

- First item
- Second item
- Third item

A paragraph for reference.

Figure 7: Bullets are positioned to the left of the text.

So, to align the *bullets* to the left you can now set a margin on the list items to line things up:

```
ul {  
    margin-left: 0;  
    padding-left: 0;  
}  
  
ul li {  
    margin-left: 1em;  
}
```

...at this point you will still find a pixel-level difference between browsers but the effect is basically as consistent as possible—see Figure 8:

A paragraph for reference.

- First item
- Second item
- Third item

A paragraph for reference.

Figure 8: Bullets positioned together with the surrounding paragraphs.

Ordered lists

Now you need to consider the same issue as applied to ordered lists. They are trickier since the numeric markers are aligned according to the list item with the largest number. For example, if you have 10 list items, decimals will be positioned to allow for the two-digit "10" item, as seen in Figure 9:

Figure 9: The numeric marker for items 1-9 have preceding padding so they right-align with item 10.

So, there really isn't a way to make this consistently left-aligned to the same position as the surrounding text; unless you set the list to use `list-style-type: decimal-leading-zero;`, which will hide the issue, as seen in Figure 10:

1. First item
2. Second item
3. Third item
4. Fourth item
5. Fifth item
6. Sixth item
7. Seventh item
8. Eighth item
9. Ninth item
10. Tenth item

01. First item
02. Second item
03. Third item
04. Fourth item
05. Fifth item
06. Sixth item
07. Seventh item
08. Eighth item
09. Ninth item
10. Tenth item

Figure 10: The leading zeros fill in the space for items 1-9.

It is more common to simply live with the difference in spacing. It does however mean that the markers of your ordered and unordered lists can't easily be consistently left-aligned. You can only line up the *text* of your lists:

```
ul, ol {  
    margin-left: 0;  
    padding-left: 0;  
}  
  
li {  
    margin-left: 2em;  
}
```

You need *at least* 2em of left margin to accomodate both ordered and unordered lists. In Figure 11, note the way the text of the items lines up in both lists:

A paragraph for reference.

- First item
- Second item
- Third item
- Fourth item
- Fifth item
- Sixth item
- Seventh item
- Eighth item
- Ninth item
- Tenth item

A paragraph for reference.

1. First item
2. Second item
3. Third item
4. Fourth item
5. Fifth item
6. Sixth item
7. Seventh item
8. Eighth item
9. Ninth item
10. Tenth item

A paragraph for reference.

Figure 11: The text lines up in both ordered and unordered lists.

So, what to do?

You basically have three choices:

1. Live with the default positioning of lists and their markers
2. Explicitly line up the text of your lists
3. Set a different style for `ul` and `ol`.

There is no "right" or "wrong" approach and it is quite common to simply leave things to the default settings for lists in general content.

Using list-style-position

If you want the text of multi-line list items to wrap below the list marker, you will need to set the `list-style-position` property to `inside`, which produces the result seen in Figure 12:

- First item - Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus quis ipsum. Quisque eget tortor mattis nunc laoreet tempus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Vivamus quam. In varius justo ultricies dolor. Duis nec pede sed dui vehicula tincidunt.
- Second item - Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus quis ipsum. Quisque eget tortor mattis nunc laoreet tempus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Vivamus quam. In varius justo ultricies dolor. Duis nec pede sed dui vehicula tincidunt.
- Third item - Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus quis ipsum. Quisque eget tortor mattis nunc laoreet tempus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Vivamus quam. In varius justo ultricies dolor. Duis nec pede sed dui vehicula tincidunt.

Figure 12: List position "inside" causes the text to wrap below the marker instead of in line with the indented text.

Inside positioned markers is not an especially popular style. By default `list-style-position` is set to `outside`, which produces the results discussed elsewhere in this article.

What about definition lists?

In general, definition lists don't require a huge amount of attention, except to set a `dt` style (commonly bold text) and control the indentation of the definitions:

```
dt {  
  font-weight: bold;  
}  
  
dd {  
  margin-left: 2em;  
}
```

This sets up a clear and easy style for definition lists, as seen in Figure 13:

Term

Definition of the term.

Term

Definition of the term.

Term

Definition of the term.

Figure 13: A simply styled definition list.

Although definition lists can be rearranged with floats and positioning, they are quirky and generally it's better to keep things simple. They are useful enough as they are, just with a little help to make the definition terms stand out more; and to get the definitions to indent nicely.

Nested lists

In the [HTML Lists article](#) you learned about nesting lists. When you create your CSS you should be careful to maintain clear design cues to show the relationship between a nested list and the list that contains it. By far the most common way to do this is by indenting the nested list items—it is in fact the default setting across the browsers.

If you set up your own list indentation, your base setting will simply be multiplied. For example, consider this CSS:

```
ul, ol {  
    margin-left: 0;  
    padding-left: 0;  
}  
  
li {  
    margin-left: 2em;  
}
```

Each subsequent child list item in the chain inherits the `margin` value from its parent list item, in addition to having another 2em of its own added on top. So a top level list item (one that doesn't have a list item as a parent element) will have a left margin of 2em, then a child list item of the first list item will inherit 2em from its parent, and then have another 2em added on to it, for a total of 4em ... and so on.

Horizontal lists

One of the most common changes required to work with a list is to produce a horizontal list—that is, to make the items appear next to each other instead of one after the other. This is a common trick for site navigation. Let's use an example from the navigation menus article (see Figure 14):

- [Home](#)
- [About Us](#)
- [Our Clients](#)
- [Our Products](#)
- [Our Services](#)
- [Contact Us](#)

Figure 14: A simple list.

Let's convert this to a horizontal list, as shown in Figure 15:

[Home](#) [About Us](#) [Our Clients](#) [Our Products](#) [Our Services](#) [Contact Us](#)

Figure 15: A simple horizontal list.

To achieve this, we need to do three things to our list:

1. Remove the `margin` and `padding` from the ``
2. Set the list items to `display: inline;`
3. Give the list items some spacing to the right, to avoid having them run together

In the example the list has the ID "mainmenu" so we'll use that as a contextual selector, to make sure we only change the list we intend to change. The CSS is:

```
#mainmenu {  
    margin: 0;  
    padding: 0;  
}  
  
#mainmenu li {  
    display: inline;  
    padding: 0 1em 0 0;  
}
```

In this simple example, setting the list items to `display: inline;` is enough; be aware that using `float: left;` will also achieve a similar look. You will [learn more about floats](#) later on in the course.

Faux columns

Earlier on we created a list of RSS feeds. Now let's imagine that list has been placed in a sidebar on your site. The designer wants the list to appear in two columns, with a border around the whole group as seen in Figure 16.

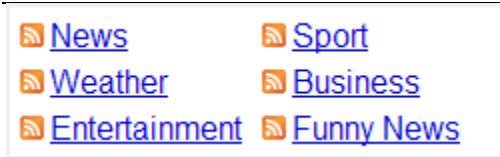


Figure 16: A list of feeds in two columns, with an RSS icon for each bullet.

Let's assume the list is inside a `<div>` which sets the width and border. The basic list would look something like Figure 17:

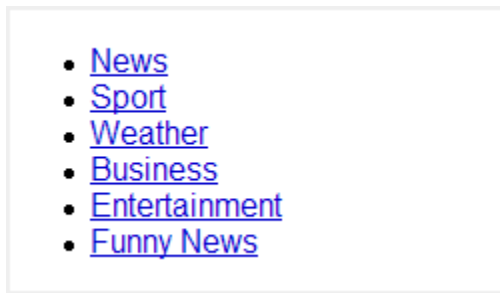


Figure 17: The unstyled list inside the border.

First, add the RSS icon as demonstrated earlier; then add 5px margin to the top, right and left:

```
.rss {  
  margin: 5px 5px 0 5px;  
  padding: 0;  
}  
  
.rss li {  
  list-style-type: none;  
  background: #fff url("icon-rssfeed.gif") 0 3px no-repeat;  
  padding: 0 0 5px 15px;  
}
```

We don't need to add bottom margin as the last list item will add the correct spacing with its padding, as seen in Figure 18:

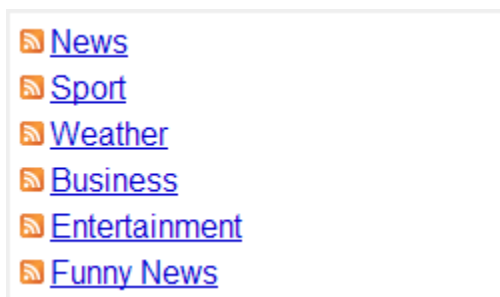


Figure 18: Halfway there—we now have correct spacing and icon bullets.

Now set the list items to `display: inline-block;` and set their width to 40% and right margin to 2% (you could also use pixel widths). You also need to explicitly set the `` to have 100% width, to ensure that the list wraps and sizes correctly:

```
.rss {  
  margin: 5px 5px 0 5px;  
  padding: 0;  
  width: 100%;  
}
```

```
}  
  
.rss li {  
  display: inline-block;  
  width: 40%;  
  margin: 0 2% 0 0;  
  list-style-type: none;  
  background: #fff url("icon-rssfeed.gif") 0 3px no-repeat;  
  padding: 0 0 5px 15px;  
}
```

In most browsers this will be enough to create the column effect, but you will need to explicitly set IE to float the list items to the left. Let's use a conditional style for all versions up to IE7 (since we don't yet know what future versions will do):

```
<!--[if lte IE 7]>  
  <style type="text/css">  
    .rss li {  
      float: left;  
    }  
  </style>  
<![endif]-->
```

We now have the desired two column effect, as seen in Figure 19:

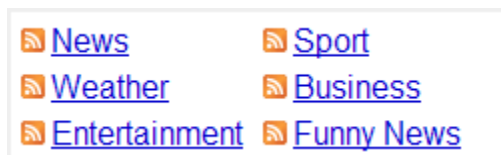


Figure 19: The completed list.

Legacy browsers

If you are required to produce this design for older browsers that don't support inline-block, then you will need to float the list items to the left in all browsers and use a clearing fix like the one described in the article [Clearing a float container without source markup](#). Thankfully the latest round of browser releases have made inline-block a viable display property, so unless you have a very large browser share for older browsers such as Firefox 2 you should be able to use the inline-block method.

Lists conclusion

We've now covered a core set of styling choices and methods for lists. You can build on these examples and combine them to create a large number of designs. Since lists are very commonly combined with links, let's move on to links.

Styling Links

Styling links can be a bit of an art form. There are many different requirements at play and it can be hard to accommodate them all, while still creating an aesthetically pleasing result. That said, it is quite possible so long as you keep some simple rules in mind:

- understand the different link states
- do not stray too far from user expectations
- use colours carefully

If you follow those rules you should produce links that are clear and easy to use.

Understanding link states

Before you can style links, you need to understand the different link states. There are five states in total: unvisited/default, visited, focus, hover and active.

unvisited

The default state of a link when it has not been activated or visited previously.

visited

The state of a link the user has already visited.

focus

Applies while the link has focus—for example while a keyboard user's cursor is on that link. Note: IE does not currently support the focus state, and just uses `active` in place of `focus`.

hover

Applies while a user is "hovering" over the link with a pointer like a mouse, but has not yet clicked the link.

active

Applies while the user activates the link—literally *during* the time they are clicking it. In some browsers this style also applies when the link has been opened in another window or tab.

You should always specify CSS for every one of these states. Each one conveys information to the user about the fact they are interacting with a link. If in doubt about `focus`, `hover` and `active` you can simply style `focus` and `hover` in the same way as their functions are similar enough that the same link style should not actively cause confusion. You can then add some simple variation for `active`, for example setting the text to italics. At a pinch you can style all three the same way.

Note that these states are not all mutually exclusive (although it is not really possible for a link to be unvisited and visited at the same time)—it is however perfectly possible for a link to be hovered, active and visited at the same time.

How browser evolution set expectations

To better understand some common user expectations about links, it helps to know a little web history.

You may hear people refer to the "Netscape defaults" for links; or say that links should always be blue and purple. This harks back to the very early days of the web, when browsers set the colours for content and authors didn't have much control over the rendering of their pages.

Text was black; the background was grey; and all links were underlined. The unvisited links were blue, visited links were purple and active links were red...and that was about it. See Figure 20 for an illustration of this.



Figure 20: A screenshot of Mosaic.

While this did get a bit monotonous it was *consistent*—and it set the baseline for user expectations. In particular, to this day users expect underlined text to be a link. They may not expect all links to have underlines, but they definitely expect underlined text to be clickable. It is best not to clash with that expectation.

Some sites still use blue and purple links; and those link colours are still the default for unstyled content in most browsers. While you can always go retro and stick with this colour set, users are generally quite comfortable with other options—within certain boundaries.

User expectations

There are some general rules for user expectations regarding links:

- Users expect links to look different from other text which is not a link
- Users expect links to react when they hover or focus on the link
- Users expect links to change after they have visited that link
- Users like consistency in link styles of the same functionality so they know what to click
- Users expect underlined text to be a link—so don't use underlines for anything else

You should always cater to these basic rules, as they will help your users quickly identify and use links. You want to create styles that don't make anyone stop and think "which bits are links?!"

These expectations translate to some simple coding rules:

- set styles for all link states
- only use underlining for links

Use colour carefully

When you are styling links, be careful not to rely entirely on colour to distinguish between link states. Not everyone can see colour the same (eg people with colour blindness), so you should use colour *and* styles like different underlines, icons or reversed colours.

You should also check that your colour choices have enough contrast—this is really easy using tools like the [Colour Contrast Analyser \(for both PC and Mac\)](#) or the [Web Accessibility Toolbar for Opera](#) (both from the Paciello Group).

The Colour Contrast Analyser (see Figure 20) allows you to use a colour picker to select the foreground and background colours on screen, then receive a simple evaluation of their contrast:

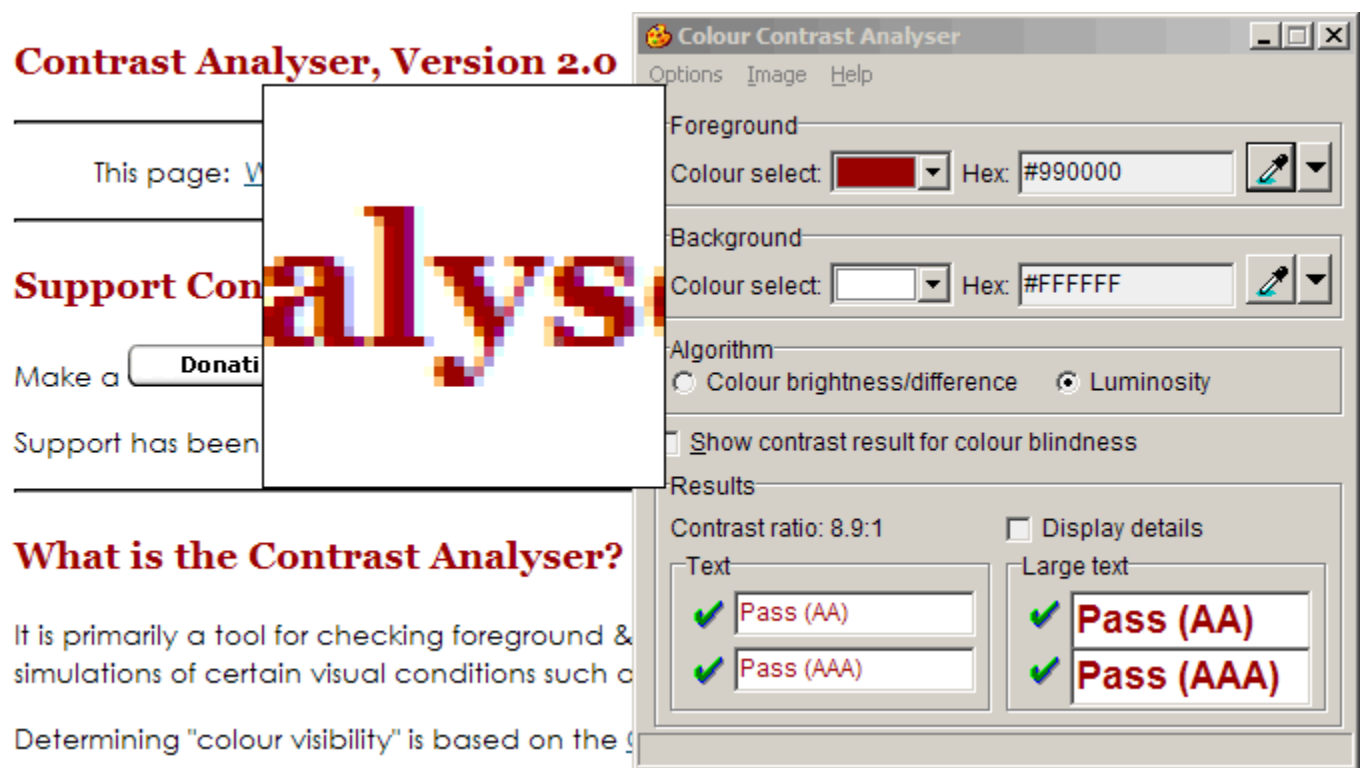


Figure 21: Screenshot of the Colour Contrast Analyser in use.

If all four results show a pass, the colour combination is ok. Remember to check all link states. You may need to enter some of them manually in the "hex" field to check focus, hover and active.

Getting down to business: the CSS

Now that you understand some ground rules for links, let's get into some code—this section details all the CSS you'll need to successfully style links.

Styling link states in the right order

First off, be aware that if you do not place your links styles in the right order in your stylesheet, the settings will override each other and the link states won't work. Your link styles must always be in the following order:

1. link
2. visited
3. focus
4. hover
5. active

A common mnemonic for remembering this is "Lord Vader's Former Handle, Anakin". If you're not a Star Wars fan, I'm afraid you'll just have to remember it the hard way, or copy and paste the code block below!

Also popular is the mnemonic "LoVe Fears HATe", with "Fears" standing for "Focus".

The different link states are styled using their "pseudo classes"—`:link` `:visited` `:focus` `:hover` `:active`—which you append to the link element selector, `a`. So your starting CSS should look like this:

```
a:link{}  
a:visited{}  
a:focus{}  
a:hover{}  
a:active{}
```

If you want to set a CSS rule for all links in all states, you can style `a` directly. Just remember to place the generic rule first, to preserve the order:

```
a {}  
a:link{}  
a:visited{}  
a:focus{}  
a:hover{}  
a:active{}
```

This is useful if you are planning to replace the default underline with a bottom border, which is a common way to gain better visual control of the style.

Controlling defaults

By default, most browsers set all links to have an underline and focus state links to have an outline, as illustrated in Figure 22:



Figure 22: Left to right: the default focus styles for Opera 9, Firefox 2 and IE7.

If you are replacing these styles with something else, you can change or disable these defaults.

Underlining

Underlining is set using the [text-decoration](#) property:

```
a {  
  text-decoration: underline;  
}
```

You can disable the underline by setting the property to `none`:

```
a {  
  text-decoration: none;  
}
```

Even if you are conserving the underline style, you may find it easier to disable `text-decoration` and use `border-bottom` to set a faux underline. We will go through this in the example below.

Outline

The focus outline is controlled by the [outline](#) property. Outline is much the same as border, but it does not take up space or cause the page to re-flow when it appears (note that it is not supported in IE7 and below either). The easiest way to control outline is with the shorthand property:

```
a:focus{  
  outline: thick solid #000;  
}
```

This example would be rendered something like Figure 23:



Figure 23: example rendering of a thick black outline.

If you are in doubt about what to do with the outline, simply leave the outline to the browser default.

Example: recreating the Netscape defaults

As an easy example of link styles, let's recreate the Netscape defaults of blue, purple and red. We'll keep the underline, but extend the active state to use italics. We'll increase the text size for the sake of the example and set the page to use a white background:

```
body {  
  background: #fff;  
  color: #000;  
  font-size: 2em;  
}  
  
a {  
  text-decoration: underline;  
}  
  
a:link{  
  color: #0000CC;  
}  
  
a:visited{  
  color: #6D006D;  
}  
  
a:focus{  
  color: #CC0000;  
}  
  
a:hover{  
  color: #CC0000;  
}
```

```
a:active{
  color: #CC0000;
  font-style: italic;
}
```

This should produce something like Figure 24:

[link](#), [visited](#), [focus](#), [hover](#), [active](#)

Figure 24: recreating the Netscape defaults.

Faux underlines using border-bottom

Many designers have observed that underlining is a bit thick and cuts through descenders of lowercase type—that is, the line goes through the bottom of g, j, p, q and y. This is illustrated in figure 25:

[pygmy](#)

Figure 25: The underline cuts through lowercase type descenders.

Let's assume that the person designing your site agrees, and would like the underline to be thinner and not touch the text. To carry out this common request, we'll use a border instead of an underline so it looks like Figure 26:

[pygmy](#)

Figure 26: Using a border instead of an underline gives nicer results.

First, disable the underline for all link states, then set a bottom-border to match the link colour for each state:

```
body {
  background: #fff;
  color: #000;
  font-size: 2em;
}

a {
  text-decoration: none;
}

a:link{
  color: #00c;
  border-bottom: 1px solid #00c;
}

a:visited{
  color: #6D006D;
  border-bottom: 1px solid #6D006D;
}

a:focus{
  color: #c00;
  border-bottom: 1px solid #c00;
}

a:hover{
  color: #c00;
  border-bottom: 1px solid #c00;
}
```

```
a:active{
  color: #c00;
  border-bottom: 1px solid #c00;
  font-style: italic;
}
```

This should produce something like Figure 27:

link, visited, focus, hover, active

Figure 27: The faux-underline in action.

If you do use the faux border method, be careful that you have sufficient `line-height` set to avoid the underline clashing with the next row of text.

Styles that don't rely on colour

Since the example so far relies purely on colour to distinguish four of the five link states, we should take the next step and change the bottom border for visited, focus and hover. Let's give visited links a dotted border, and hover and active a dashed border:

```
body {
  background: #fff;
  color: #000;
  font-size: 2em;
}

a {
  text-decoration: none;
}

a:link{
  color: #00c;
  border-bottom: 1px solid #00c;
}

a:visited{
  color: #6D006D;
  border-bottom: 1px dotted #6D006D;
}

a:focus{
  color: #c00;
  border-bottom: 1px dashed #c00;
}

a:hover{
  color: #c00;
  border-bottom: 1px dashed #c00;
}

a:active{
  color: #c00;
  border-bottom: 1px solid #c00;
  font-style: italic;
}
```

This should produce something like Figure 28:

link, visited, focus, hover, active

Figure 28: Changing the border style for each link state.

Accepting focus and hover as equivalent styled states, this method means the link states are distinguished with more than colour. Even if you were to view these links in black and white, you could identify the different link states, as shown in Figure 29:

link, visited, focus, hover, active

Figure 29: The link states are now distinguishable even in black and white.

Icons on links

Some sites use icons and symbols to add information about their links. For example, some sites use an arrow to indicate that a link goes to an external site; or they use a tick to show the link has been visited.

These effects are simple to achieve with background images, as shown in Figure 30:

external link ↗, visited link ✓

Figure 30: An example of links with distinguishing icons.

To add an arrow icon to external links you could add the class "external" to the link tag:

```
<a href="http://example.com/" class="external">external link</a>
```

Then in your stylesheet, set a background image for that class—remembering to add padding to accommodate the image:

```
a.external {  
  background: #fff url("icon-external.gif") center right no-repeat;  
  padding-right: 30px;  
}
```

This example would apply the icon to all instances of visited links, in all states. If you wanted to restrict the icon just to *unvisited* external links, you can combine classes and the link state pseudo classes in your selector:

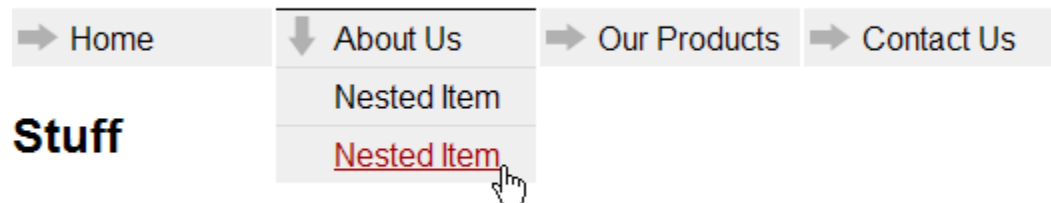
```
a.external:link{  
  background: #fff url("icon-external.gif") center right no-repeat;  
  padding-right: 30px;  
}
```

Combining classes and states opens up a wide range of creative possibilities for your links. Remembering to check your colours, your only limitation from this point is creativity.

Bringing it all together—a simple navigation menu

To illustrate one way to bring lists and links together, the examples zip includes a [simple flyout navigation menu](#), as seen in Figure 31. Flyout menus are a very common navigation system.

Home



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis eu leo. Pellentesque massa. Phasellus ut sem luctus arcu ullamcorper aliquet. Mauris turpis turpis, posuere non, cursus a, blandit vel, massa. Nunc nisl urna, sodales quis, consectetur nec, rutrum

Figure 31: Screenshot of the example flyout menu.

Summary

A good grasp of styling lists and links is essential for web developers, as they are used everywhere. They are routinely combined to create site navigation; while a clear link style is critical for the ease of use of any site. Bad link styles can be seriously confusing for everyone and may even make a site unusable for some people.

Exercise questions

- How do you choose between basic list styles, for example square bullets or Roman Numerals on an ordered list?
- What is an image sprite and why would you use one?
- Why is colour contrast important and how do you make sure your link colours are using suitable colours?
- What is the correct order for setting styles on the different link states?

Further reading

- [WCAG Samurai Errata for WCAG 1.0](#), with specific reference to [Guideline 2. Don't rely on colour alone](#).
 - [Type and Colour \(a chapter from Building Accessible Websites, by Joe Clark\)](#)
 - [Juicy Studio: Highlighting Links](#)
 - [Max Design—Simple, accessible external links](#)
 - [Resource Center—Contrast Analyser 2.0 \(Paciello Group\)](#)
 - [A List Apart: CSS Sprites: Image Slicing's Kiss of Death](#)
- [Previous article—CSS background images](#)

About the author



Ben Buchanan started creating web pages more than ten years ago, while completing a degree in everything but IT. He has worked in both the public (university) and private sectors; and worked on the redevelopment of major websites including [The Australian](#) and three generations of [Griffith University's](#)

corporate website. He now works as Frontend Architect for [News Digital Media](#) and writes at [the 200ok weblog](#).

33: Styling tables

BY BEN BUCHANAN · 26 SEP, 2008

Published in: [BACKGROUND](#), [ZEBRA](#), [BUG](#), [BORDERS](#), [ALIGN](#)

This is Article 33 of the Opera Web Standards Curriculum

[Previous article—Styling lists and links](#)

[Next article—Styling forms](#)

[Table of contents](#)

Introduction

At times it seems that tables are a little misunderstood in modern web development. So much attention is given to "don't use tables!" that people sometimes forget the issue is actually "don't use tables *for layout*". Tables are excellent for their true purpose—displaying tabular data. So it makes sense to know how to style them properly.

This tutorial will focus on applying CSS in an efficient manner, to produce clear and readable data table styles. I'll also cover some common design requests for tables. The article structure is as follows:

- [Table structure](#)
- [The basics](#)
 - [Table and cell width](#)
 - [Text alignment](#)
 - [Borders](#)
 - [Padding](#)
 - [Caption placement](#)
 - [Backgrounds](#)
 - [Fixing IE with conditional styles](#)
- [Common variations](#)
 - [Zebra striping](#)
 - [Uneven columns](#)
 - [Incomplete grids](#)
 - [Inner grids](#)
- [Two common bugs](#)
 - [Border-collapse bug](#)
 - [Margin/caption bug](#)
- [Summary](#)
- [Exercise questions](#)
- [Futher reading](#)

You may find it useful to [download the code examples for tables shown in this article](#), so you can follow along with the article as it progresses.

Table structure

Before we dive into the CSS, let's consider the key structural elements of tables you will need to style clearly:

- Table headings
- Table data cells
- Table captions

When your site users read your table, they should be able to easily understand and follow the structure of the table. The most common way to do this is with borders, background colours, or both.

You do not have to follow these style conventions, however, you should ensure that there is some clear difference between `th` and `td` cells; also, the `caption` should be clearly associated with the table and differentiated from other text on the page.

The basics

Consider the way this unstyled table is rendered (this is the same example you met in [Article 19—HTML tables](#)):

Volcano Name	Location	Last Major Eruption	Type of Eruption
Compiled in 2008 by Ms Jen			
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption

The data is understandable, but it does takes some effort to work out what's happening. Let's add some style to make it easier to read.

Table and cell width

The first decision is how wide to make the table. The browser default is the same as setting `table { width: auto; }`, which results in the table extending to the width of the content. This generally looks untidy.

Let's imagine that our table is going into a content column 600px wide. Let's set the table to expand to 100% of the available width, to make best use of space. Since there are four columns, let's also set the width of the table cells to an equal 25% each:

```
table {  
  width: 100%;  
}  
  
th, td {  
  width: 25%;  
}
```

You can actually just set the width on `th` and it will set the width of all the columns; however, it doesn't hurt to be thorough. This simple style will produce the result seen in Figure 1:

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 1: The example table with simple width settings.

The cells are now sitting at an even width. We'll look at setting uneven widths later, but for now let's push on.

Text alignment

The table is still a bit confusing to read, so let's set up the text alignment to be a little neater—the additional rule below will left-align the headers to match the content (by default, browsers centre table headings):

```
table {  
  width: 100%;  
}  
  
th, td {  
  width: 25%;  
  text-align: left;  
}
```

This neatens things up a little, as you can see in Figure 2:

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 2: Table with left alignment applied.

Currently all of the cells are vertically aligned to the centre. If you prefer, you can set this to align text to the top or bottom of the cell, or in fact any [vertical-align setting](#) that you like. The new rules below set the text to align to the top:

```
table {  
  width: 100%;  
}  
  
th, td {  
  width: 25%;  
  text-align: left;  
  vertical-align: top;  
}
```

The table now looks like Figure 3:

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 3: Table with vertical alignment added.

Note how the top row of headings all sit at the top, even though "Last Major Eruption" wraps on to two lines.

Borders

The table is looking a little nicer, however it is still a bit hard to read along each line. It's time to set some borders to make things easier to read. You need to set borders separately for each part of the table, then decide how those borders should combine.

To show where the borders will be set, Figure 4 shows different borders for table (solid black), caption (solid grey), th (dashed blue) and td (dotted red):

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 4: illustration of the different element borders within a table.

Note how the `table` border runs around the outside of all the heading and data cells, then between the cells and the caption. You can also see that by default, the `th` and `td` borders are spaced out from each other.

Let's look at a different style of table - you can set up a simple black border for the table and cells, using the `border` property—this is done via the new rules below:

```
table {  
  width: 100%;  
  border: 1px solid #000;  
}  
  
th, td {  
  width: 25%;  
  text-align: left;  
  vertical-align: top;  
  border: 1px solid #000;  
}
```

Which produces the result seen in Figure 4:

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 4: Table with simple black borders applied.

This makes the rows far easier to read, however you may not like the spacing between the cells. There are two ways to change this.

First, you can simply close the gaps using the `border-spacing` property, like so:

```
table {
  width: 100%;
  border: 1px solid #000;
}

th, td {
  width: 25%;
  text-align: left;
  vertical-align: top;
  border: 1px solid #000;
  border-spacing: 0;
}
```

This will make the borders touch together instead of sitting apart. This changes the 1px border into a 2px border, as seen in Figure 5:

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 5: Table with border spacing removed, producing a 2px border effect.

You can also increase the space between cells using `border-spacing`, although bear in mind that this property doesn't work in Internet Explorer.

If you want to retain the 1px border effect, you'll need to set the table so that the borders "collapse" into each other. You can do this using the `border-collapse` property instead of the `border-spacing` property:

```
table {
  width: 100%;
  border: 1px solid #000;
}
```



```
}  
  
th, td {  
  width: 25%;  
  text-align: left;  
  vertical-align: top;  
  border: 1px solid #000;  
  border-collapse: collapse;  
}
```

This will produce a table with a 1px border, like in Figure 6:

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 6: Table with border-collapse set to collapse, reducing the border to 1px.

When you set borders to collapse, you need to keep in mind that this can cause issues if you have different border styles applied to adjacent cells. When the different border styles are collapsed into each other, they will "conflict" with each other. This is resolved according to the [W3C CSS2 Table border conflict resolution rules](#), which determine which style "wins" when they are collapsed.

Padding

Now that you have borders on the cells, you might like to add some whitespace to the caption and table cells. You simply use padding to accomplish this:

```
table {  
  width: 100%;  
  border: 1px solid #000;  
}  
  
th, td {  
  width: 25%;  
  text-align: left;  
  vertical-align: top;  
  border: 1px solid #000;  
  border-collapse: collapse;  
  padding: 0.3em;  
}  
  
caption {  
  padding: 0.3em;  
}
```

This allows the text to "breathe" a little, as seen in Figure 7:

Recent Major Volcanic Eruptions in the Pacific Northwest

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 7: Table with padding applied to all cells.

Caption placement

So far the caption has been left sitting at the top of the table. However, you might like to move the caption somewhere else. Unfortunately you cannot do this in IE, but for all other browsers you can change the position of the caption using the `caption-side` property. The options are top, bottom, left and right. Let's move the caption to the bottom:

```
table {
  width: 100%;
  border: 1px solid #000;
}

th, td {
  width: 25%;
  text-align: left;
  vertical-align: top;
  border: 1px solid #000;
  border-collapse: collapse;
  padding: 0.3em;
  caption-side: bottom;
}

caption {
  padding: 0.3em;
}
```

Figure 8 shows the result.

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Recent Major Volcanic Eruptions in the Pacific Northwest

Figure 8: Table with caption moved to the bottom of the table.

If you do want move the caption, remember that any side-specific styles will not work in IE. For example if you add three borders to make the caption "join" the bottom of the table, it won't have the desired effect in IE because the caption will still be at the top. You will need to use [conditional comments](#) to re-style your table for IE. See also the [Fixing IE with conditional styles](#) section later on, for more details.

For the rest of the examples, I will leave it at the top.

Backgrounds

Another simple way to style tables is to add background colours and images. This is done with the `background` property, although you need to be aware that the different parts of the table will "layer" over each other. [The CSS2 specification explains background layering in some detail](#) however the short version is that backgrounds will override each other in the following order:

1. table (which sits at the "bottom" or the "back")
2. column groups
3. columns
4. row groups
5. rows
6. cells ("top" or "front", meaning their background overrides all the others)

So, if you set a background for the table, and a different colour for cells, the cell background will cover up the table background. If you have borders set to `collapse`, the table background won't show at all. If you set `border-collapse` to `separate`, however, the table background will show through between the borders.

Note that the concept of different elements sitting on top of one another on the page is controllable; you can control how high or low in the "stack" an element sits in relation to other elements by changing its `z-index` property. You will [learn more about z-index](#) in Article 37.

Imagine you set the table to have a red background and cells to have a white background. Separated cells will show the red, but the cells stay white, as demonstrated by Figure 9:

Recent Major Volcanic Eruptions in the Pacific Northwest

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 9: Table demonstrating the red table element background showing between the white backgrounds of the cell elements.

You can also use a background image. For example, if you wanted to have a gradient showing through between the cells, you could set the `th` and `td` cells to white backgrounds, but set the `table` background to a gradient:

```
table {
  border-collapse: separate;
  border-spacing: 5px;
```

```
background: #000 url("gradient.gif") bottom left repeat-x;
color: #fff;
}

td, th {
background: #fff;
color: #000;
}
```

Note that the background colour is set to black, which will fill up the space at the top where the gradient graphic finishes (you should always allow for your table being taller than the background image). The foreground colour is set to white, in case these default colours ever show through to the cell content. In general, the cell styles will override the text colour settings from the `table {}` style, but you should always declare contrasting foreground and background colours at each level.

These styles produce a table which would look like Figure 10 in most browsers:

Recent Major Volcanic Eruptions in the Pacific Northwest

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 10: Table demonstrating a gradient background image showing through between the cells.

By default IE won't show as much of the background, since it doesn't support `border-spacing`. However you will still get the same general effect, as indicated by Figure 11.

Recent Major Volcanic Eruptions in the Pacific Northwest

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 11: The smaller border-spacing gap rendered by IE.

Depending on your circumstances, you may be happy to simply accept this different rendering between browsers. Of course that isn't always an option, for example when a client particularly wants a design to look exactly the same in all browsers.

Fixing IE with conditional styles

There is a workaround for the IE problems listed above. It requires a hack and an extra stylesheet, but it works. You can use an `expression` to produce the wider gap, then load that expression using conditional comments. The expression syntax is:

```
table {  
  border-collapse: expression("separate", cellSpacing = "5px");  
}
```

This CSS is only useful to IE, so you only want IE to apply it. The expression will also invalidate your stylesheet, so many developers prefer to isolate IE hacks in a separate stylesheet only loaded by IE.

To do this, create a new stylesheet named `ie-only.css` and link it within conditional comments:

```
<!--[if lte IE 7]><link rel="stylesheet" media="screen" href="ie-only.css"  
</><![endif]-->
```

Note the `[if lte IE 7]` means "if less than or equal to IE version 7". This reveals the code to IE7 and all earlier versions of IE, while the surrounding HTML comment hides the code from all other browsers. You can adjust this to whichever version of IE you need to target, for example to target IE6 and earlier use `[if IE 6]`.

In your main stylesheet, set the normal style:

```
table {  
  border: 1px solid #000;  
  border-collapse: separate;  
  border-spacing: 5px;  
  background: #000 url("gradient.gif") bottom left repeat-x;  
}
```

Then set your IE-only style in `ie-only.css`:

```
table {  
  border-collapse: expression("separate", cellSpacing = "5px");  
}
```

This will get IE to produce a table with wide cell spacing. You just have to remember to maintain the extra width settings—if you update your main stylesheet, you will have to update `ie-only.css` as well. Obviously conditional comments allow you to do a lot more than just style tables, since the extra stylesheet can contain as much CSS as you need to fix IE bugs.

A simple design

Most designs use relatively simple combinations of backgrounds. Let's give the table headers a grey background, and change the caption to be white text on black:

```
table {  
  width: 100%;  
  border: 1px solid #000;  
}  
  
th, td {  
  width: 25%;  
  text-align: left;  
  vertical-align: top;  
  border: 1px solid #000;  
  border-collapse: collapse;  
  padding: 0.3em;  
  caption-side: bottom;  
}
```

```
caption {
  padding: 0.3em;
  color: #fff;
  background: #000;
}

th {
  background: #eee;
}
```

This looks like Figure 12:

Recent Major Volcanic Eruptions in the Pacific Northwest			
Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 12: Table with reversed white-on-black caption and light grey background applied to the table heading cells.

Common variations

In this section I will look at some common design archetypes you will see again and again in tables across the Web.

Zebra striping

A common design request for tables is to create rows with alternating colours. These are commonly referred to as "zebra striping". Although there is [some conjecture as to whether zebra striping actually helps the reader](#), they are a popular style. Figure 13 shows an example:

Common table elements		
Element	Tag	Purpose
table	<table>	Encloses a table
table row	<tr>	Encloses a row of table cells
table header cell	<th>	Sets a heading for a row or column
table data cell	<td>	Contains data
caption	<caption>	Sets a caption for the table

Figure 13: A table with "zebra stripes", alternate rows set to white or light grey.

The simplest way to accomplish zebra stripes is to add a class to alternate table rows, then use a contextual CSS selector to style the cells in those rows. First, the classes "odd" and "even" are added to the table rows, like so:

```
...  
<tr class="odd">  
...  
<tr class="even">  
...
```

You can skip the heading row as it already has its own style. You then add a contextual class to set the background for all cells inside odd class rows:

```
.odd th, .odd td {  
  background: #eee;  
}
```

This is the simplest way to add zebra striping to an HTML table that will work across all browsers, but it is not perfect—what if you add a row to the table? You'd then need to move all your odd and even class names around to get everything looking right again.

There are two other options:

- You can add the classes using unobtrusive JavaScript, as demonstrated in [A List Apart: Zebra Tables](#). Most JavaScript frameworks have a suitable method, too: [Zebra Table Showdown](#) compares a range of framework implementations.
- You can use the CSS3 `:nthchild` selector, however, this isn't supported across all the major browsers yet. Support will improve as time goes on though.

You can [find more out about zebra striping with nth-child in a dedicated dev.opera.com article](#).

Incomplete grids

Some designs will respond well to less structured, more open-looking grids. A simple variation is to remove the vertical borders and leave out the background fill on the caption, as seen in Figure 14:

Recent Major Volcanic Eruptions in the Pacific Northwest

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption
Compiled in 2008 by Ms Jen			

Figure 14: A table with lighter grey borders only on the outer edge and bottom of each cell.

The CSS for this effect is:

```
table {
```



```

width: 100%;
border: 1px solid #999;
text-align: left;
border-collapse: collapse;
margin: 0 0 1em 0;
caption-side: top;
}

caption, td, th {
padding: 0.3em;
}

th, td {
border-bottom: 1px solid #999;
width: 25%;
}

caption {
font-weight: bold;
font-style: italic;
}

```

You can take this a step further and remove all of the borders, except a top and bottom border to give some definition to the table body—see Figure 15:

Recent Major Volcanic Eruptions in the Pacific Northwest

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption

Compiled in 2008 by Ms Jen

Figure 15: A table with borders applied only to the top and bottom of the table body.

The CSS for this effect is:

```

table {
width: 100%;
text-align: left;
border-collapse: collapse;
margin: 0 0 1em 0;
caption-side: top;
}

caption, td, th {
padding: 0.3em;
}

tbody {
border-top: 1px solid #000;
border-bottom: 1px solid #000;
}

tbody th, tfoot th {
border: 0;
}

```

```

th.name {
  width: 25%;
}

th.location {
  width: 20%;
}

th.lasteruption {
  width: 30%;
}

th.eruptiontype {
  width: 25%;
}

tfoot {
  text-align: center;
  color: #555;
  font-size: 0.8em;
}

```

Inner grids

Sometimes you will want to remove the outer border, but keep the inner grid of borders, like in Figure 16:

Recent Major Volcanic Eruptions in the Pacific Northwest

Volcano Name	Location	Last Major Eruption	Type of Eruption
Mt. Lassen	California	1914-17	Explosive Eruption
Mt. Hood	Oregon	1790s	Pyroclastic flows and Mudflows
Mt. St. Helens	Washington	1980	Explosive Eruption

Compiled in 2008 by Ms Jen

Figure 16: A table with an inner grid design.

To accomplish this for all current browsers, you need to add a class to the `th` and `td` cells that appear last on each row, like this:

```

...
<tr>
  <th scope="col">Volcano Name</th>
  <th scope="col">Location</th>
  <th scope="col">Last Major Eruption</th>
  <th scope="col" class="last">Type of Eruption</th>
</tr>
...

```

Then we use that class to remove the right border from those cells. The full CSS is:

```

table {
  width: 100%;
  text-align: left;
  border-collapse: collapse;
}

```

```
margin: 0 0 1em 0;
caption-side: top;
}

caption, td, th {
padding: 0.3em;
}

th, td {
border-bottom: 1px solid #000;
border-right: 1px solid #000;
}

th.last, td.last {
border-right: 0;
}

tfoot th, tfoot td {
border-bottom: 0;
text-align: center;
}

th {
width: 25%;
}
```

Inner grids using `:lastchild`

When browser support improves, we will be able to use the pseudo selector `:lastchild` to achieve this effect without classes. The CSS would be:

```
table {
width: 100%;
text-align: left;
border-collapse: collapse;
margin: 0 0 1em 0;
caption-side: top;
}

caption, td, th {
padding: 0.3em;
}

th, td {
border-bottom: 1px solid #000;
border-right: 1px solid #000;
}

th:lastchild, td:lastchild {
border-right: 0;
}

th {
width: 25%;
}
```

This currently works in the latest versions of Opera, Firefox and Safari.

Two common bugs

In this last section I'll cover two really common bugs, so you're prepared for when they crop up. They concern borders and captions.

border-collapse bug

When you set your table to `border-collapse: collapse;` you will find that Firefox and Safari will incorrectly display the width of table features. For example, if you set a 1px border on the table, cells and caption, Firefox will render the caption 1px too narrow on the left, as seen in Figure 17:

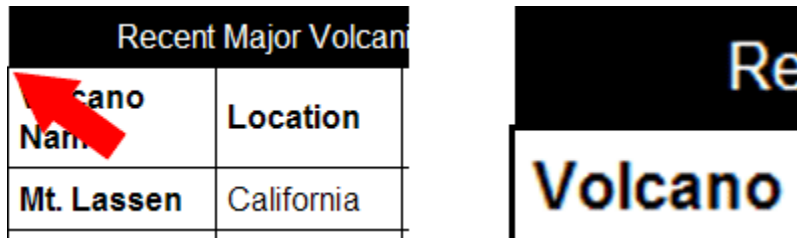


Figure 17: The border-collapse bug affects Firefox and Safari.

Safari does the same thing, just on the right. This bug is all based on a rounding issue that ultimately comes down to how you display "0.5 of a pixel". It can be argued that this is not a bug per se, but the browsers don't agree so it's effectively a bug.

So what's the solution? If you want to use a 1px border and a caption background, there really isn't a fix other than to "live with it". It is a very minor difference and a non-fatal problem—that is, the table remains entirely usable. So, many people choose to just live with the differences between browsers. Let the Web be the Web.

If you are happy to use a larger border, say 2px, then you can simply set a 1px border on table, cells and caption; then set table to separate borders and apply zero spacing between them:

```
table {
  border-collapse: separate;
  border-spacing: 0;
  border: 1px solid #000;
}

th, td, caption {
  border: 1px solid #000;
}
```

In Firefox at least, the 1px borders will add up to the desired 2px rendered border, avoiding the rounding problem on the way. Safari still leaves a gap.

Alternatively, you can hide the problem by not using a border or background colour on your caption. The problem is still there; you just won't see it. This is probably the simplest and most effective solution.

Margin/caption bug

If you use a caption and set a margin on `table`, you need to be aware that Firefox and Safari may place the table margin *between* the table cells and the caption.

To combat this in Firefox, you can set the margin on three sides of `table`, set the `caption-side` explicitly, then add the fourth margin to the `caption`. Unfortunately, this solution will invoke the bug in Safari. So, this isn't really a fix unless you are willing to live with the bug in either Firefox or Safari.

The only way to avoid a problem in both Firefox *and* Safari is to set a zero margin on the side with the caption. For example, if your caption is at the top you could just set your margin on the right, bottom and left sides; or just the bottom. This may work if you set all of your margins on the same side of content elements, so the margin isn't required to space the table from adjacent content.

Summary

By now you should have a good grasp of the fundamental styling options available for tables. There are some limitations imposed by browser inconsistencies, but in general you should be able to create clear and readable tables without any trouble. Just pay attention to your borders, give the text some breathing room, and be careful with backgrounds.

Exercise questions

- How do you control the space between table and cell borders?
- What happens when `table` has one background colour, `th` and `td` cells have another background colour, and `border-collapse` is set to `collapse`?
- How do you set different columns to have different widths?

Further reading

- [W3C: CSS2 Tables](#), with particular reference to the [CSS2 table background layering](#) section.
- [A List Apart: A Dao of Web Design](#)—"let the web be the web". A timeless article which will explain why a 1px difference between browsers doesn't truly matter.
- [A List Apart: Zebra Tables](#) and [A List Apart: Zebra Striping: Does it Really Help?](#)
- [Zebra striping tables with CSS3](#)
- [Supporting IE with conditional comments](#)
- [A CSS styled table | Veerle's blog](#) & [A CSS styled calendar | Veerle's blog](#)
- [Data Tables and Cascading Style Sheets Gallery](#) shows off a variety of table designs (although be aware many do not meet [W3C colour contrast recommendations](#)).

About the author



Ben Buchanan started creating web pages more than ten years ago, while completing a degree in everything but IT. He has worked in both the public (university) and private sectors; and worked on the redevelopment of major websites including [The Australian](#) and three generations of [Griffith University's](#) corporate website. He now works as Frontend Architect for [News Digital Media](#) and writes at [the 200ok weblog](#).

34: Styling forms

BY [BEN HENICK](#) · 26 SEP, 2008

Placeholder. Article will follow very soon!

35: Floats and clearing

BY [TOMMY OLSSON](#) · 26 SEP, 2008

Introduction

In this article you will get acquainted with floating and clearing—two must-have tools for the modern web designer. They are versatile tools that you can use to allow text to flow around images or even create multi-column layouts.

The structure of this article is as follows:

- [What are float and clear for?](#)
- [Some boring theory](#)
- [How does floating work?](#)
 - [The minutiae](#)
 - [More floats](#)
 - [Margins on floats](#)
- [Clearing](#)
- [Containing floats](#)
- [Shrink-wrapping](#)
- [Centering floats](#)
- [Bugs!](#)
- [Summary](#)
- [Exercise questions](#)

What are float and clear for?

If you look in a typical magazine you'll see images illustrating the articles, with the text flowing around them. The `float` property in CSS was created to allow this style of layout on web pages. *Floating* an image—or any other element for that matter—pushes it to one side and lets the text flow on the other side. *Clearing* a floated element means pushing it down, if necessary, to prevent it from appearing next to the float.

Although floating was intended for use with any elements, designers most commonly use it to achieve multi-column layouts without having to abuse table markup.

Some boring theory

In order to explain how floating works, you need to peek under the bonnet and look at how a web browser renders an HTML/CSS document. Don't worry, I'll be brief.

Each visible HTML element generates a *box* which is then rendered. If you're viewing the document on a computer screen or a mobile phone, the boxes are rendered on the display. If you're printing the document, the boxes are rendered on paper. If you're using a screen reader, the content of the boxes is rendered aurally, as speech.

Just as there are block-level and inline elements in HTML, there are block-level and inline boxes in CSS. By default, block-level elements generate block-level boxes and inline elements generate inline boxes. There will also be some generated boxes in addition to the ones generated by elements, for instance, for the text content of the document. Block boxes are normally laid out in the order the elements appear in the markup, from top to bottom. Block boxes cannot appear side-by-side unless we apply some CSS. Inline boxes are laid out horizontally. The `direction` property determines if they're laid out from left to right or from right to left (the default is left to right, if this is not specified)

This is known as the *document flow*: inline boxes flow horizontally within their parent block boxes, and block boxes flow vertically. The boxes occur in the same order as the elements in the HTML markup.

Consider the following simple HTML document (I've only included the part inside the `body` element):

```
<p>This is a very simple document.</p>
<p>It consists of <em>two</em> paragraphs.</p>
```

Figure 1 shows a screen shot of that document with an overlay that shows the two block boxes generated by the `p` elements and the inline box generated by the `em` element.

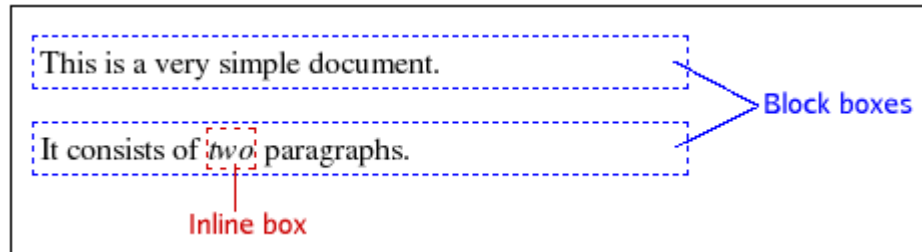


Figure 1: A demonstration of block boxes generated by the `p` elements, and an inline box generated by the `em` element.

All the inline boxes that make up one “line” on the output device are enclosed in imaginary rectangles known as *line boxes*. Line boxes are always laid out from the top down with no space between them, as illustrated in Figure 2.

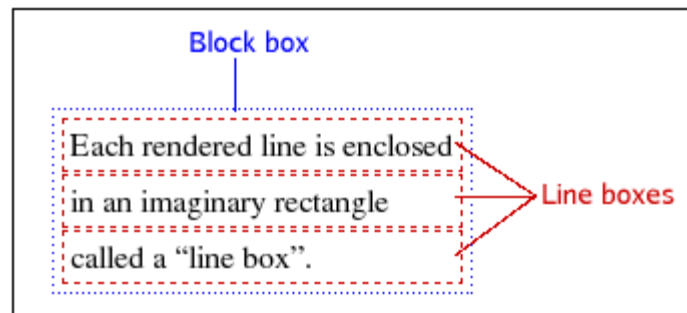


Figure 2: Each rendered line is enclosed in a separate line box.

How does floating work?

OK! Now that we've gone through all the boring theoretical stuff, let's move on to look at the syntax of floats and clearing, and check out some examples.

The `float` property has four valid values: `left`, `right`, `none` and `inherit`. The first two are by far the most commonly used and will cause a box to be floated to the left or to the right. The declaration `float:none` which is the default, is normally only declared to “undo” a declaration in some other rule. The use of `float:inherit` is probably very rare—I've never seen it used in the wild—and exists probably just for the sake of consistency. It would make the element inherit the `float` value from its parent element.

A floated box is taken out of the document flow and shifted as far as possible to the left or to the right, depending on the specified floating direction. “As far as possible” usually means until the outer edge of the float touches the edge of the containing block (the inside of its padding, if any). Thus, for `float:left` the box is moved to the left until the left margin of the float touches the left edge of the parent.

The alert reader may have noticed that I said “usually” above. If there is already a box floated to the left when we float another box in the same direction, the second box will stop when it touches the first box. In other words, floats don't climb on top of one another.

It's time to look at floating in action, so get your text editor ready.

1. Create a new file, copy the code below into it, and save the document as `float.html`.

```

2.      <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"      http://www.w3.org/TR/html4/strict.dtd">
3.      <html>
4.          <head>
5.              <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
6.              <title>Floating</title>
7.          </head>
8.          <body>
9.              <p id="p1"><span id="span-a">Lorem ipsum</span>
10.                 <span id="span-b">dolor sit amet</span>
11.                 <span id="span-c">consectetuer</span> adipiscing elit.
12.                 Curabitur feugiat feugiat purus.
13.                 Aenean eu metus. Nulla facilisi.
14.                 Pellentesque quis justo vel massa suscipit sagittis.
15.                 Class aptent taciti sociosqu ad litora torquent per conubia nostra,
per inceptos hymenaeos.
16.                 Quisque mollis, justo vel rhoncus aliquam, urna tortor varius
lacus, ut tincidunt metus arcu vel lorem.
17.                 Praesent metus orci, adipiscing eget, fermentum ut, pellentesque
non, dui.
18.                 Sed sagittis, metus a semper dictum, sem libero sagittis nunc,
vitae adipiscing leo neque vitae tellus.
19.                 Duis quis orci quis nisl nonummy dapibus.
20.                 Etiam ante. Phasellus imperdiet arcu at odio.
21.                 In hac habitasse platea dictumst. Aenean metus.
22.                 Quisque a nibh. Morbi mattis ullamcorper ipsum.
23.                 Nullam odio urna, feugiat sed, bibendum sed, vulputate in, magna.
24.                 Nulla tortor justo, convallis iaculis, porta condimentum, interdum
nec, arcu.
25.                 Proin lectus purus, vehicula et, cursus ut, nonummy et, diam.</p>
26.
27.                 <p id="p2">Nunc ac elit. Vestibulum placerat dictum nibh. Proin
massa.
28.                 Curabitur at lectus egestas quam interdum mollis.
29.                 Cras id velit a lacus sollicitudin faucibus.
30.                 Proin at ante id nisi porttitor scelerisque.
31.                 In metus. Aenean nonummy semper enim.
32.                 Aenean tristique neque quis arcu tincidunt auctor.
33.                 Fusce consequat auctor ligula.
34.                 Fusce nulla lorem, sagittis a, lacinia et, nonummy in, eros.
35.                 In nisi augue, aliquam eget, convallis vel, malesuada quis,
libero.</p>
36.
37.                 <p id="p3">Hello, World!</p>
38.          </body>
      </html>

```

That's a lot of content, but we need some to show how this works.

39. Open the document in your web browser to see how it looks. Boring, isn't it?
40. Create another document in your text editor, populate it with the code below, and save it as `style.css` in the same directory as the HTML file from Step 1.

```

41.      #span-a {
42.          float: left;
43.          background-color: #cfc;
44.          color: #030;

```

```
}
```

45. Link the style sheet to the HTML document by inserting the following line just before the `</head>` tag:

```
<link rel="stylesheet" type="text/css" href="style.css">
```

46. Save and refresh the page in your browser. You'll now see the `span` element containing the words "Lorem ipsum" floated to the left. I've also given it a light green background, to make it stand out a bit.
47. It's still not easy to see what's happening here, so let's make our float a little larger. Add the following declaration to your style sheet:

```
48. #span-a {  
49.     float: left;  
50.     background-color: #cfc;  
51.     color: #030;  
52.     padding: 1em;  
}
```

53. Save and refresh, and you'll see that the green area is now larger, since we've added a bit of padding on all four sides of the box. The float is as tall as three lines of text and we can clearly see that the other text is flowing around the float.

The minutiae

Now I'll analyse what's happening here in further detail. The floated box generated by the first `span` element has been shifted to the left, all the way to the edge of the document, and the line boxes adjacent to it have been shortened. Although it's not readily visible yet, the block box generated by the paragraph that contains the float is not affected. Let's highlight the paragraph to make this clearer.

1. Add another CSS rule to the style sheet, as follows:

```
2. p {  
3.     border: 1px solid #f00;  
}
```

4. Again, save the CSS file and to refresh the browser. You should now see a red border around each paragraph—notice that the float resides inside one of the paragraphs.
5. Let's modify the last rule, to verify that floats stop at the inner edge of the parent's padding area:

```
6. p {  
7.     border: 1px solid #f00;  
8.     padding: 1em;  
9.     background-color: #ff9;  
}
```

10. Save and refresh, and you'll see proof of what I said earlier: the floated box is shifted to the edge of its containing block, while the parent's padding lies outside it. You'll also see that the yellow background of the paragraph extends underneath the floated box. Floating a child box clearly isn't affecting the paragraph box, only the line boxes within.
11. Let's experiment some more—what happens if the float is taller than its parent? Modify the rule for the float as follows:

```
12.    #span-a {  
13.        float: left;  
14.        background-color: #cfc;  
15.        color: #030;  
16.        padding: 1em 1em 10em;  
  
    }
```

Note: If you've got a narrow browser window you may need to use a larger value than 10em for the bottom padding to get the green area to extend past the bottom border of the paragraph.

You will now see something interesting: the floated box protrudes outside the parent block; the parent box does not expand to contain its floating child box. You can also see (if you've used a large enough bottom padding) that the line boxes adjacent to the float in the *second* paragraph are shortened.

More floats

Let's create another float to see what happens when two elements are floated in the same direction.

1. Add a new rule to your style sheet and save and refresh as before:

```
2.    #span-b {  
3.        float: left;  
4.        background-color: #ccf;  
5.        color: #003;  
6.        padding: 1em;  
  
    }
```

Now the span element containing the words "dolor sit amet" are also floated to the left. You will see that it's shifted to the left until it touches the first float; in other words, "as far as possible".

7. Why stop at two floats? Let's make a third—add the following rule to your style sheet:

```
8.    #span-c {  
9.        float: left;  
10.       background-color: #fcc;  
11.       color: #300;  
12.       padding: 2em 1em;  
  
    }
```

13. I also want you to add a temporary rule to see an example of what happens when there isn't enough room for a float on a line. Add the following rule at the end of the style sheet:

```
14.    span {  
15.        width: 34%;  
  
    }
```

16. As before, save your style sheet and refresh the document in the browser—you'll see something like the output shown in Figure 3.

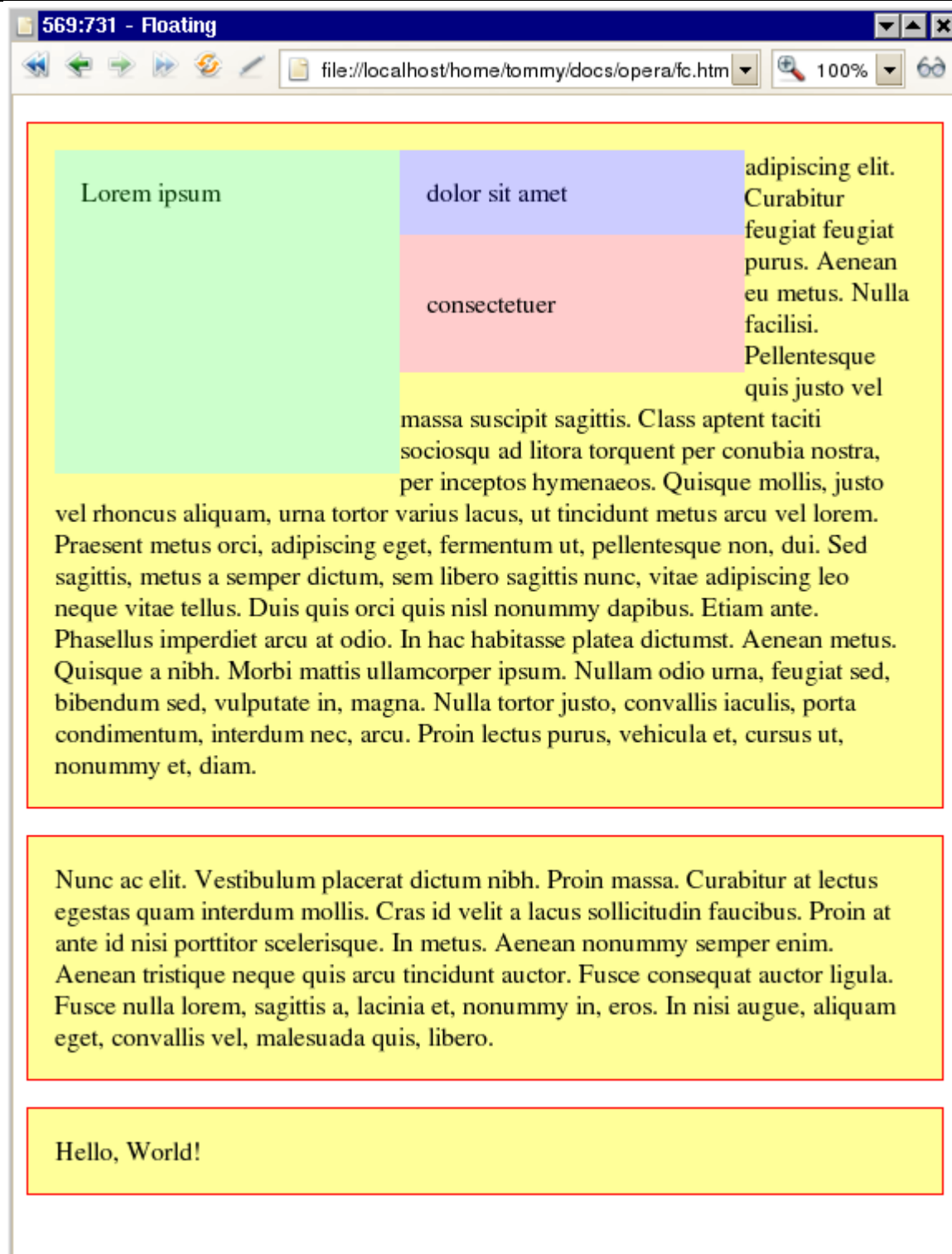


Figure 3: Not quite what you expected?

Whoa! What just happened? The third float now appears *below* the second one! (And Internet Explorer 6 does some other strange things, which we'll ignore for now.) Since the width of each `span` element is 34% of the paragraph's width (as specified by the rule added in Step 3), plus some padding, there isn't room for all three of them side-by-side ($3 \times 34\% = 102\%$). The first two floats fit on the same line, but the third one

does not and is shifted down. The important thing is that it's only shifted down as far as it needs to be, to fit within the line. It's not shifted down below the first, tall, float since there's room to the right of it.

Another interesting thing to note here is that you've assigned a width to the `span` elements. This shouldn't make any difference since `span` is an inline element type. Floating a box, however, automatically makes it a block-level box, which means we can assign dimensions and vertical margins to it.

Margins on floats

Now we'll explore what you can do with margins on floats.

1. First, remove the temporary rule for `span` elements that you added earlier, and then save and refresh, so that our three floats exist side-by-side again. In other words, delete this rule:

```
2.     span {  
3.         width: 34%;  
  
    }
```

Now the floats are stacked tightly together and the adjacent text starts immediately after the last float (unless you are using Microsoft Internet Explorer 6 or older, in which case there's a 3-pixel gap on the right due to [the three pixel jog bug](#)). How can you make some space around a floated box? The answer is margins!

4. Let's try this on the middle float—change the CSS rule for the middle float as follows, then save and refresh:

```
5.     #span-b {  
6.         float: left;  
7.         background-color: #ccf;  
8.         color: #003;  
9.         padding: 1em;  
10.        margin-left: 1em;  
11.        margin-right: 1em;  
  
    }
```

Yep, now there's some space on both sides of the middle float.

12. You can also set vertical margins on a floated box—make the following changes to the rule for the third float, then save and refresh.

```
13.     #span-c {  
14.         float: left;  
15.         background-color: #fcc;  
16.         color: #300;  
17.         padding: 2em 1em;  
18.        margin-top: 2em;  
19.        margin-bottom: 2em;  
  
    }
```

This makes the third float move down and there's also some extra space below it.

20. Since we're in an adventurous mood, let's see what happens if we start playing with *negative* margins! Make the following changes to the rule for the third float, then save and refresh:

```
21.     #span-c {  
22.         float: left;  
23.         background-color: #fcc;
```

```

24.     color: #300;
25.     padding: 2em 1em;
26.     margin-top: 2em;
27.     margin-bottom: 2em;
28.     margin-left: -4em;

```

}

You'll now see the output shown in Figure 4.

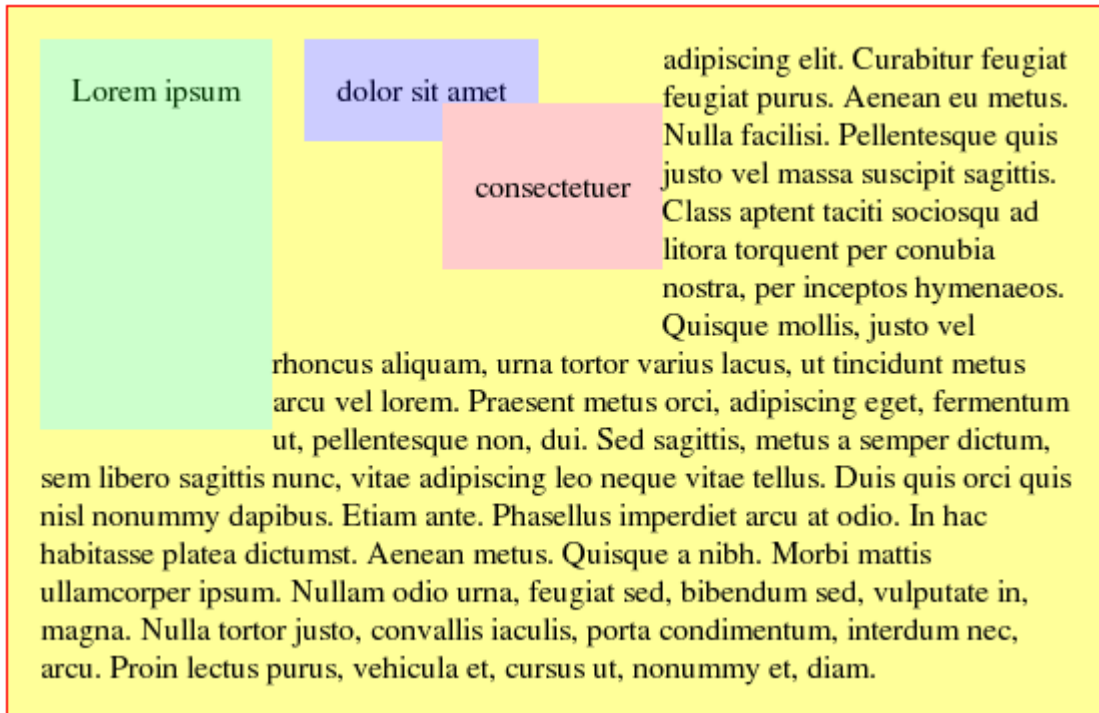


Figure 4: You'll now see floats on top of one another!

How about that, eh? Who said floats can't appear on top of other floats? Note how the negative left margin moves the entire float to the left.

Using negative margins on floats can be very useful in certain types of multi-column layouts.

Clearing

Now that I've covered the basics of floating, I'll move on to the closely related topic of clearing.

As you have seen in the examples throughout this article, text will flow around a floated element, and block boxes aren't affected by floats. Sometimes it's desirable to make sure that an element doesn't end up adjacent to a float. For instance, a heading that introduces a new section of an article shouldn't appear next to an image from the previous section. You'd much rather have the heading appear below the image, even if the image protrudes below the last paragraph. The only way to do that is to use the `clear` property on the heading.

Another example is the ubiquitous three-column layout with a full-width footer. If the columns are floated, you use the `clear` property on the footer to ensure that it appears below all the columns—no matter which column happens to be the longest.

The `clear` property has three useful values: `left`, `right` and `both`. The values `none` (default) and `inherit` are also valid.

Using `clear:left` on an element means that its generated box is guaranteed to appear below any previously floated boxes on the left side. If you use `clear:both` it will appear below all previous floats on either side.

Clearing is achieved by shifting the element down (white space is added above its top margin) if necessary, until its top edge is below the bottom edges of all floated boxes in the specified direction(s). Let's look at an example to illustrate it further.

1. Before you try this, let's clean up your style sheet. Remove the rules for `#span-b` and `#span-c` so that you only have the green float left. Make sure that its bottom padding is large enough that it extends into the second paragraph.
2. Add the following rule for the second paragraph, then save and refresh:

```
3.     #p2 {  
4.         clear: left;  
  
     }
```

Observe! The second paragraph is shifted down until it clears free of the float, as seen in Figure 5.

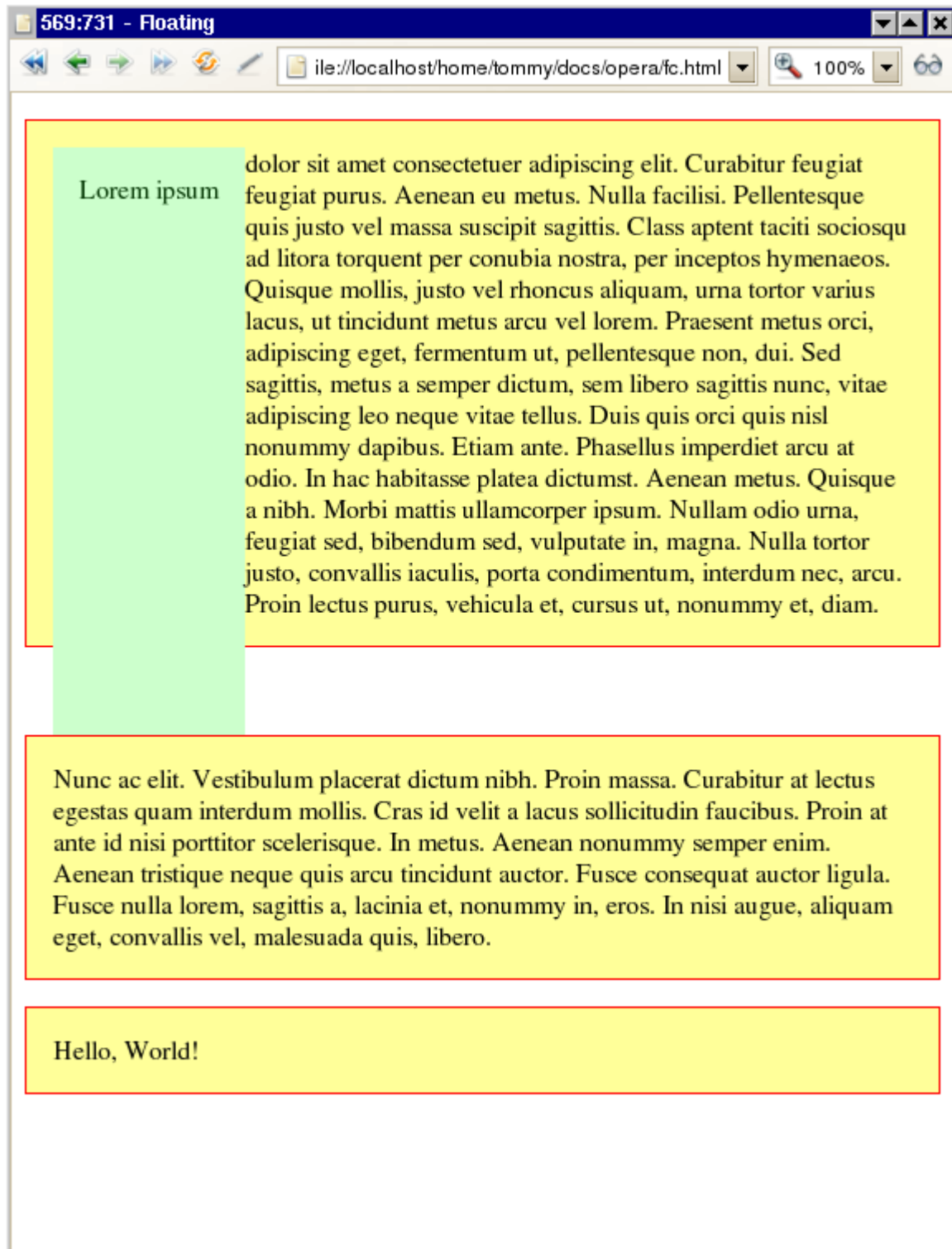


Figure 5: The second paragraph is now cleared below the first one.

To make things really complicated we can use `float` and `clear` on the same element.

5. Add a rule for the second float and let it clear the first float, then save and refresh:

```
6.      #span-b {  
7.          float: left;  
8.          clear: left;  
9.          padding: 1em;  
10.         background-color: #ccf;  
11.         color: #003;
```

```
}
```

The blue float now appears *below* the green float, entirely outside the parent paragraph. Since it's also floated to the left, the second paragraph is shifted down even further to clear it.

Containing floats

As you've seen above, the parent box doesn't normally expand to contain floated children. This can often cause confusion, for example when *all* children of an element are floated when you make a horizontal menu out of an unordered list by floating all the `li` elements. Since floated boxes are taken out of the flow and don't affect the parent box, floating all the children effectively makes the parent empty and it will collapse to zero height. Sometimes this is undesirable, for instance if you want to set a background on the parent. If the parent has zero height, no background will be visible.

It's obvious that we need some mechanism for making a parent box expand to enclose its floated children. The traditional method was to include an extra element in the markup, just before the parent's closing tag, and setting `clear: both` on it. That works, but it's rather unpalatable since it involves introducing extra unnecessary, unsemantic markup. Fortunately, there are other ways, which I'll discuss now.

The first method is simply to float the parent, too. Floated boxes will always expand to enclose their floated children.

1. To try it in our example document, remove the rule for `#span-b` again, and float the first paragraph like so, before saving and refreshing:

```
2.      #p1 {  
3.          float: left;
```

```
}
```

The paragraph now expands until it encompasses the green float. This is all well and good, but sometimes floating the parent isn't an option. Another way that avoids floating the parent is to set the `overflow` property of the parent to something other than `visible`. If you set it to `hidden` and don't specify a height, the parent will enclose floated children.

4. Replace the last rule with this, then save and refresh:

```
5.      #p1 {  
6.          overflow: hidden;
```

```
}
```

Note that the latter method doesn't work in Internet Explorer 6 or older.

Shrink-wrapping

I mentioned earlier that floating an inline box caused it to become block-level, thus allowing us to specify dimensions and vertical margins for it. Floating a *block* box also has a surprising consequence: if no width is specified the box will "shrink-wrap" to fit its content. This wasn't visible in the example document when you floated the first paragraph, because it had enough content to fill the whole window (unless you have a *really* wide monitor).

Let's float the last paragraph to see the effect. In fact, just to have some variation, let's go really wild and float this one to the right!

Add the following rule to the style sheet, then save and refresh:

```
#p3 {  
    float: right;  
}
```

The paragraph that says, "Hello, World!" will now be floated to the right and is only as wide as the text plus some padding that you specified in a previous rule for all paragraphs.

Centering Floats

Sometimes you will want to float an element—perhaps to make it enclose floated children—while having it horizontally centred within its parent. There is a problem here: you can't use the usual trick of setting the left and right margins to `auto` for floats, and there is no such value as `float:center`. Isn't there some way to work around this?

There is, as a matter of fact. CSS guru Paul O'Brien explains how in his article [When is a float not a float?](#). It involves an extra wrapper element, but you can live with that. The principle uses relative positioning, which we will cover in the next article, [CSS static and relative positioning](#). By shifting the wrapper element to the right, then shifting the float back to the left, you can actually center a shrink-wrapped float of unknown width! (You can use this to impress your partner on the next date. It never fails.)

Let's try it. In the following example you'll add a horizontal menu bar to your page, based on an unordered list with floating items.

1. Insert the following markup just after the `<body>` tag in your HTML document:

```
2.     <div class="wrap">  
3.         <ul id="menu">  
4.             <li><a href="#">Home</a></li>  
5.             <li><a href="#">News</a></li>  
6.             <li><a href="#">Products</a></li>  
7.             <li><a href="#">Services</a></li>  
8.         </ul>  
9.     </div>  
10.  
11.     <!--Internet Explorer needs this-->
```

```
<div class="clear"></div>
```

12. Add the following CSS rules to your style sheet, to style the menu:

```
13.     #menu {  
14.         margin: 0;  
15.         padding: 0.5em;  
16.         font-family: Verdana,sans-serif;  
17.     }  
18.  
19.     #menu li {  
20.         float: left;  
21.         list-style-type: none;  
22.         margin: 0 0 0 0.5em;  
23.         padding: 0.25em;  
24.         background-color: #600;  
25.         color: #ff9;  
26.         border: 2px solid #f00;  
27.     }  
28.
```

```
29.     #menu a {
30.         color: #ff9;
31.         text-decoration: none;
32.     }
33.
34.     .wrap {
35.         float: left;
36.         margin-bottom: 2em;
37.     }
38.
39.     .clear {
40.         clear: left;
41.         height: 1px;
42.         margin-top: -1px;
```

```
}
```

43. Save both files and refresh the browser. You'll see your menu in the top left-hand corner. Let's make it horizontally centred.
44. Shift the wrapper element halfway across the page, by modifying the rule for `.wrap` as shown below:

```
45.     .wrap {
46.         float: left;
47.         margin-bottom: 2em;
48.         position: relative;
49.         left: 50%;
```

```
}
```

Your menu will start at the horizontal centre of the page, but that's not what we wanted—it's too far to the right, so you need to shift it back a bit to the left. Since you have floated the wrapper, it's shrink-wrapped to fit the list. You want to move the list a distance that equals half of its width, which also means half the width of the wrapper, so you shift it by `-50%`.

50. Modify the `#menu` rule as follows:

```
51.     #menu {
52.         margin: 0;
53.         padding: 0.5em;
54.         font-family: Verdana,sans-serif;
55.         position: relative;
56.         left: -50%;
```

```
}
```

The menu is now centred; the only problem is that you may have a horizontal scroll bar, depending on the width of the list and the browser window. This is because you have shifted the wrapper element halfway across the screen; if the list is wider than half the window, part of it will end up off-screen.

57. You can get rid of that by setting `overflow:hidden` on a suitable parent element, to make the overflow hidden. In this case the parent of the wrapper is the `body`. Sometimes it's not feasible to hide overflow on the `body` element, in which case you'll need a wrapper for the wrapper; in this case, however, it's fine.

Add the following rule to your style sheet:

```
body {
    overflow: hidden;
```

```
}
```

58. Actually, there is one more problem. If you look at this in Internet Explorer you'll see that it still isn't working properly. The workaround is to float the list itself, but only in Internet Explorer, since it breaks other browsers. You can get round this by using a little hack that ensures that only Internet Explorer applies the rule.

Add the following rule to your style sheet:

```
* html #menu {  
    float: left;  
}
```

Bugs!

Floating and clearing is very useful, but unfortunately most—if not all—browsers have buggy implementations of these properties. Internet Explorer 6 boasts an astounding array of odd behaviour with floats, including disappearing content, doubled margins and the infamous 3-pixel jog. But not even Firefox and Opera are completely bug-free when it comes to floating and clearing. [Position Is Everything](#) is an invaluable resource where these bugs are documented—along with workarounds in most cases.

Summary

Floating a box shifts it as far as it can go to the left or to the right inside its parent element. A floated box is taken out of the document flow and doesn't affect the parent box or subsequent block-level boxes, although adjacent line boxes are shortened. When there isn't room for a floated box on a line because of previous floats, it is shifted down until it fits (or until there are no other floats).

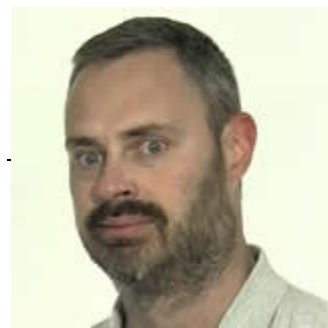
When an inline box is floated it becomes a block-level box. When a block-level box is floated and no explicit width is specified, it will shrink-wrap to fit its content.

Clearing floats entails shifting content down, if necessary, until the top edge of the content is below the bottom edges of all floating boxes in the specified direction(s).

Centering a shrink-wrapped floating box is possible by adding a wrapper element and some judicious use of relative positioning.

Exercise questions

- What happens if you float an element in the middle of a paragraph; ie, if there is text before the float? Make sure to try this in different browsers, because they behave differently. Opera and Safari get it right, while Firefox and Internet Explorer don't.
- How can floats be used to display image thumbnails in a gallery of equally sized "cells" without using a layout table?
- How can you have a vertical navigation menu on the left-hand side of the page and a content column on the right, without the content text wrapping under the menu?
- A very common web site layout consists of a full-width header, below which there are three content columns and then a full-width footer at the bottom. How can you achieve such a layout with floating and clearing?



About the author

Creative Commons Attribution, Non Commercial - Share Alike 2.5 license.
re ASA. All rights reserved.

Tommy Olsson is a pragmatic evangelist for web standards and accessibility, who lives in the outback of central Sweden. He wrote his first HTML document in 1993 and is currently the technical webmaster for a Swedish government agency.

He has written one book so far—The Ultimate CSS Reference (with Paul O’Brien)—and has a sadly neglected blog called [The Autistic Cuckoo](#).

36: CSS static and relative positioning

BY TOMMY OLSSON · 26 SEP, 2008

Published in: [INLINE](#), [LAYOUT](#), [COLUMN](#), [BLOCK](#), [IE](#)

This is Article 36 of the Opera Web Standards Curriculum

[Previous article—Floats and clearing](#)

[Next article—CSS absolute and fixed positioning](#)

[Table of contents](#)

Introduction

In this article I'll start looking in depth at how you can use CSS to position HTML elements wherever you want on the page, using the `position` CSS property and some related properties.

The `position` property in CSS has four legal values (in addition to the ubiquitous `inherit`): `static`, `relative`, `absolute` and `fixed`. These values have a significant impact on how an element is rendered. The two values `static` and `relative` are closely related, and we'll look into those in great detail in this article. The values `absolute` and `fixed` are also closely related, and I'll save those for the next article in the series.

The structure of this article is as follows:

- [The wonderful world of rectangles](#)
- [Static positioning](#)
 - [Block box layout](#)
 - [Inline box layout](#)
- [Relative positioning](#)
 - [Multi-column layout with source order requirements](#)
 - [Making columns](#)
 - [Working around quirks in Internet Explorer](#)
 - [Other uses for relative positioning](#)
- [Summary](#)
- [Exercise questions](#)

The wonderful world of rectangles

Now for a bit of a recap on CSS and HTML boxes, as discussed in [Article 35 on floats and clearing](#). An HTML document consists of a number of elements interspersed with character data (text). When such a document is rendered on a computer screen or printed on paper, those elements generate rectangular boxes. Just as the set of HTML elements is divided into block-level elements and inline elements, boxes in CSS are essentially either block boxes or inline boxes. By default, the built-in user agent style sheet in a browser makes block-level HTML elements such as `p` and `div` generate block boxes, while inline elements such as `strong` and `span` generate inline boxes. We can control the type of box that is generated using the `display` property.

The boxes generated by the elements in a document are laid out according to a clearly defined set of rules in the [CSS2.1 specification](#). Those rules are written for the relatively few people who write browser software to learn how CSS works though, not for those of us who design web pages for a living—or a hobby. This is why this entire course exists! As a result, the specification can be a bit difficult to understand. In this article I'll try to explain the basics in a way that is better suited for web designers and developers.

Static positioning

This is really a misnomer. Boxes with `position:static` are not really “positioned” at all in the CSS sense. They are simply laid out in the order they occur in the markup and take up as much room as they need—this is the default behavior you get when you don’t apply any CSS at all to your HTML.

There are fundamental differences in how block boxes are laid out compared to how inline boxes are laid out, so let’s examine the two types one at a time. I’ll start with block boxes, because they are simpler.

Block box layout

Unless we apply any specific CSS declarations, block boxes are laid out vertically from top to bottom in the order they occur in the markup. Each box is normally as wide as the document (the `body` element), but even if we make them narrower they will not be laid out side by side even if there’s room; they’ll still be laid out one below the other. You can think of it as if each block box had an implicit line break before and after it, to make sure it ends up on a “line” of its own.

The vertical distance between two block boxes is controlled by the `margin-bottom` property of the first box and the `margin-top` property of the second box (you’ve seen how to manipulate these earlier in the course). For boxes in the *normal flow*, ie boxes that aren’t floated or absolutely positioned, the vertical margins between two adjacent block boxes will *collapse*—overlap—so that the net result is not the sum of the two margins, but the greater of the two, as seen in Figure 1 below.

Consider the following HTML fragment:

```
<p style="margin-bottom:40px">This paragraph has a 40px bottom margin.</p>
<p style="margin-top:20px">This paragraph has a 20px top margin.</p>
```

When viewed in a browser, the margins collapse, as shown in Figure 1.

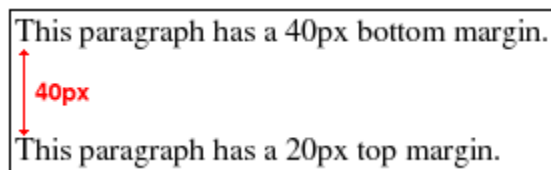


Figure 1: The margins collapse—the distance between the two is 40px, not 60px.

A block box will either contain only other block boxes or only inline boxes. If a block-level element contains a mix of block-level and inline children—which is permissible, although semantically questionable—so-called *anonymous block boxes* will be generated to encompass the inline child boxes, so that the parent only contains block boxes.

You can specify the dimensions of a block box using the `width` and `height` properties. You can also set both vertical and horizontal margins on them. The initial (default) value for `width` and `height` is `auto`, and the initial value for margin properties is 0. These factors in combination mean that a block box will by default be as wide as its parent, as illustrated in Figure 2.

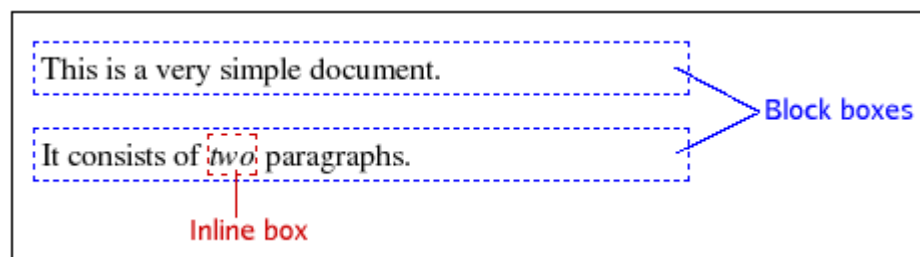


Figure 2: Block boxes are laid out vertically.

Inline box layout

This section may be difficult to understand if you're new to CSS, but don't despair if you don't get it the first time you read it. Experimenting a little on your own is probably the best way to get a solid understanding of these issues—just make sure that you're using a good, standards-compliant browser when testing, such as Opera or Firefox.

Inline boxes are generated by default by inline HTML elements, but there are also *anonymous inline boxes* generated to encompass the text content of elements.

The inline boxes are laid out horizontally, one after the other, in the order in which they occur in the markup. Depending on the `direction` property, the inline boxes will either be laid out from left to right (`direction:ltr`) or from right to left (`direction:rtl`). Left-to-right direction is used with, for instance, European languages, while right-to-left direction is used with languages such as Arabic and Hebrew.

The set of inline boxes that make up one line on the screen (or paper) are enclosed in yet another rectangle, known as a *line box*. Line boxes are laid out vertically within their block-level parent, with no space between them. We can affect the height of line boxes through the `line-height` property.

For inline boxes we cannot specify any dimensions. We can specify horizontal margins, but not vertical margins.

If necessary, an inline box will be split into several inline boxes, distributed over two or more line boxes. When such a split occurs, any horizontal margins and padding, and any vertical borders, will only apply before the first box and after the last box. Consider a document with the following rule for `em` elements:

```
em {  
  margin: 0 2em;  
  padding: 0 1em;  
  border: 1px dotted blue;  
}
```

This will give you a layout somewhat like that seen in Figure 3, when the styled elements are broken over multiple lines.

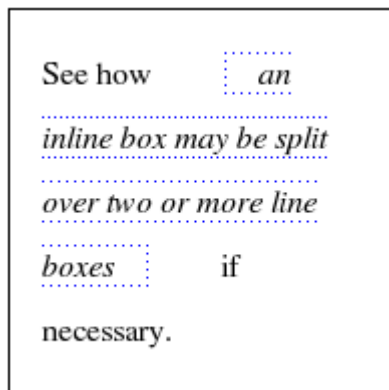


Figure 3: Margins, padding and border do not apply where breaks occur.

The vertical alignment of inline boxes within the encompassing line box is determined by the `vertical-align` property. The default value is `baseline`, which means that the inline boxes are aligned so that their text baselines line up. The baseline is the imaginary line on which letters without descenders stand. It is placed some distance above the bottom of the line box to leave room for the descenders of lowercase letters, as shown in Figure 4.

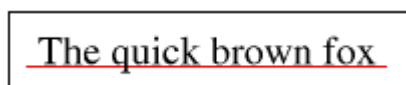


Figure 4: Letters stand on the imaginary baseline.

Note that the `vertical-align` property applies to inline boxes and table cells only, and it isn't inherited. Figure 5 shows some small images with different vertical alignment.

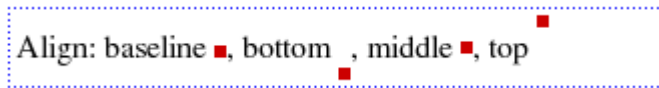


Figure 5: Images placed using settings of the `vertical-align` CSS property.

When the total width of the inline boxes within a line box is less than the width of the line box itself, the horizontal alignment is controlled by the `text-align` property. With `text-align:justify` extra space is inserted between the inline boxes, if necessary, to make the content both left- and right-justified. This property applies to block boxes, table cells and inline blocks, and it is inherited—Figure 6 shows the result of applying different values of the `text-align` property to text inside table cells.

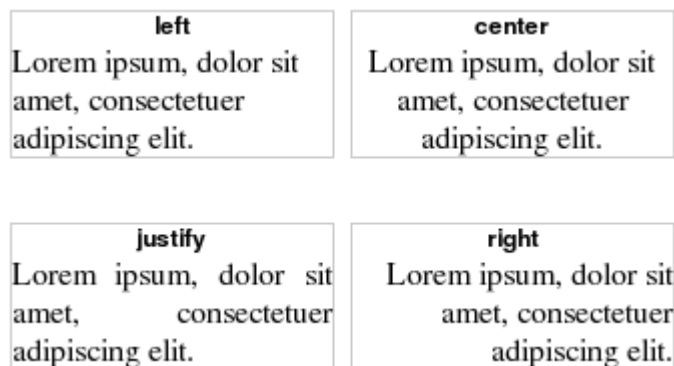


Figure 6: Controlling the alignment of text using the `text-align` property.

Relative positioning

Relative positioning is a positioning scheme in CSS, but it is more closely related to static “positioning” than with its cousins—absolute and fixed positioning.

An element with `position:relative` is first laid out just like any static element; block-level or inline. But then something interesting happens: the *generated box* is shifted according to the `top`, `bottom`, `left` and `right` properties.

The thing to remember about relative positioning is that it's only the generated box that is shifted. The element still remains where it was in the static document flow. That's where it “takes up space” as far as other elements are concerned. This means that the shifted box may end up overlapping other elements' boxes, because they still act like the relatively positioned element has remained where it should be, before the positioning was applied. As far as the document flow is concerned, the element has not moved—it is just the end visual result that shows the box being moved. Let's look at it in practice.

1. Copy the HTML code below into a new document in your favourite text editor and save it as `relative.html`.

```
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
3. <html>
4.   <head>
5.     <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
6.     <title>Relative Positioning</title>
7.   </head>
8.   <body>
```

```

9.      <p>Lorem ipsum dolor sit amet consectetur adipiscing elit.
10.     Curabitur feugiat feugiat purus.
11.     Aenean eu metus. Nulla facilisi.
12.     Pellentesque quis justo vel massa suscipit sagittis.
13.     Class aptent taciti sociosqu ad litora torquent per conubia nostra,
per inceptos hymenaeos.
14.     Quisque mollis, justo vel rhoncus aliquam, urna tortor varius lacus,
ut tincidunt metus arcu vel lorem.
15.     Praesent metus orci, adipiscing eget, fermentum ut, pellentesque non,
dui.
16.     Sed sagittis, <span>metus a semper</span> dictum, sem libero sagittis
nunc, vitae adipiscing leo neque vitae tellus.
17.     Duis quis orci quis nisl nonummy dapibus.
18.     Etiam ante. Phasellus imperdiet arcu at odio.
19.     In hac habitasse platea dictumst. Aenean metus.
20.     Quisque a nibh. Morbi mattis ullamcorper ipsum.
21.     Nullam odio urna, feugiat sed, bibendum sed, vulputate in, magna.
22.     Nulla tortor justo, convallis iaculis, porta condimentum, interdum
nec, arcu.
23.     Proin lectus purus, vehicula et, cursus ut, nonummy et, diam.</p>
24.     </body>

```

```
</html>
```

25. Open the file in your web browser to see how it looks at this stage—you should just see a plain paragraph of text.
26. Create a new document in your editor, copy the CSS code below into it and save the file as `style.css`.

```

27.    p {
28.        width: 20em;
29.    }
30.
31.    span {
32.        background-color: lime;
33.    }

```

33. Link the style sheet to the HTML document by inserting the following line just before the `</head>` tag:

```
<link rel="stylesheet" type="text/css" href="style.css">
```

34. Save both files and reload the page in your browser. I have narrowed the paragraph to make the line breaks occur at the same position even in small browser windows. The `span` element now has a migraine-inducing background colour to make it more visible.
35. Next, let's modify the style sheet by adding three declarations to the rule for the `span` element:

```

36.    span {
37.        position: relative;
38.        top: 1em;
39.        left: 2em;
40.        background-color: lime;
41.    }

```

41. Save and reload the page in the browser to see the effects of relative positioning.

You have shifted the `span` element both vertically and horizontally. Notice how it now overlaps the next line of text, and how there is an empty hole where it used to be.

The way the generated box has been shifted may not be what you expected from the code. You specified `top:1em`, but the box was shifted *downward*. Also, the box was shifted to the *right*, even though you specified `left:2em`. Why is this?

The key to understanding how these properties work with relative positioning is to realise that they specify the *edge* that the movement is applied to, not the direction of movement. In other words, the `top` property shifts the box relative to its top edge, the `left` property shifts the box relative to its left edge, and so on. The box is shifted *away* from the specified edge, so `top:1em` shifts the box 1em away from the top position—in other words, downwards. Negative numbers shift the box in the opposite direction, so `bottom:-1em` is the same as `top:1em`.

This leads us to another conclusion: it's pointless to specify both the `top` property and the `bottom` property (or `left` and `right`) for the same element. The rules of CSS say that `bottom` should be ignored if `top` is specified. For horizontal movement it depends on the `direction` property. In a left-to-right environment `right` is ignored if both `left` and `right` are specified; in a right-to-left environment `left` is ignored.

The example we just looked at explains relative positioning, but it doesn't seem very useful, does it? So what use is relative positioning? Let's look at a more involved example.

Multi-column layout with source order requirements

A word of warning: this example is a bit complex. If you're new to the world of CSS it may even appear a bit daunting, but I'll talk you through it at a gentle pace and explain what I'm doing as I go along. If you haven't yet Article 35, which covers [floats and clearing](#), now would be a good time to do so.

There's one type of layout that is very common on web sites. It consists of a page header, often containing some masthead graphic, under which there are two or more "columns" side by side. Below all this there is often a full-width footer, perhaps with a copyright statement or contact information. Figure 7 shows an example of this type of layout.



Figure 7: A typical multiple column layout, with columns sandwiched between a header and a footer.

This type of layout used to be created with layout tables back in the Dark Ages (the 1990s). That's an abuse of HTML markup for presentational purposes, which is not advised, so therefore which we will not be teaching you about in this course. CSS offers ways to achieve the same thing using `display:table-cell` and similar, but there's a major drawback to that solution: it's not currently supported by any version of Internet Explorer, so we won't look at that either. Only two options remain: floats or absolute positioning. Both methods have their advantages and drawbacks, but if you want a full-width footer and don't know in advance which column will be the longest, then floats are necessary to ensure the integrity of your design.

The problem with floats is that they only shift to the left or right until they touch the edge of the parent block, or another float. That means floated columns have to appear in the right order in your markup. But sometimes it's desirable to have a presentational order that is different from the source order. You may want to have the content before the navigation, for instance, to enhance usability for keyboard navigation and to improve search engine optimisation. This is possible to achieve, even with floats, with some judicious use of negative margins and relative positioning—let's have a look at how to do this. Let's begin with a skeleton, or wireframe, HTML document.

1. Copy the code below into your text editor and save the file as `layout.html`.

```
2.      <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">
3.      <html lang="en">
4.          <head>
5.              <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
6.              <title>Static and Relative Positioning</title>
7.              <link rel="stylesheet" type="text/css" href="layout.css">
8.          </head>
9.          <body>
10.             <div id="header">Header</div>
11.             <div id="main">Main content</div>
12.             <div id="sidebar">Sidebar</div>
13.             <div id="nav">Navigation</div>
14.             <div id="footer">Footer</div>
15.          </body>
      </html>
```

16. Next, you'll create the embryo of a style sheet. Copy the code below into your text editor and save the file as `layout.css`.

```
17.      #header {
18.          background-color: #369;
19.          color: #fff;
20.      }
21.
22.      #sidebar {
23.          background-color: #ff6;
24.      }
25.
26.      #nav {
27.          background-color: #ddd;
28.      }
29.
30.      #footer {
31.          border-top: 1px solid #369;
32.      }
```

32. Save both files and load the page in your browser. The five divisions appear in order, from top to bottom.

Imagine your design department has specified that the navigation must be on the left and the sidebar on the right, with the main content column in the middle. The header and footer should extend across the whole page width and we don't know which of the three columns in between will be the longest. The source order is mandated by your accessibility and usability experts and isn't negotiable. How can you combine all those requirements into a working layout?

You are going to have to add an extra element into the markup for this to work. It's unavoidable, but one extra element is something you ought to be able to live with. You need an element that wraps around the three "columns".

33. Insert the two highlighted lines below into the HTML document:

```
34.      <div id="header">Header</div>
35.      <div id="wrapper">
36.          <div id="main">Main content</div>
37.          <div id="sidebar">Sidebar</div>
38.          <div id="nav">Navigation</div>
```

39. `</div>`

```
<div id="footer">Footer</div>
```

The designers (who, fortunately, understand accessibility and device independence) have stipulated that the navigation needs to be 12em wide while the sidebar should be 14em. The main content column should have a fluid width, so that the layout adapts to different window sizes, since fixed-width layouts aren't very user friendly. To prevent lines of text from being too long, impeding readability, you need to constrain the layout to a maximum width. In order to prevent overlap in extremely narrow windows you also need to constrain the layout to a minimum width. Within those constraints, the layout should be centred horizontally within the browser window.

40. Next, assign the widths to the navigation and the sidebar and set the width constraints and general centering by adding the following rules to the bottom of the CSS file:

```
41.  body {
42.    margin: 0 auto;
43.    min-width: 40em;
44.    max-width: 56em;
45.  }
46.
47.  #sidebar {
48.    width: 13em;
49.    padding: 0 0.5em;
50.    background-color: #ff6;
51.  }
52.
53.  #nav {
54.    width: 11em;
55.    padding: 0 0.5em;
56.    background-color: #ddd;
57.  }
```

57. Save the files and reload—you should see that the yellow sidebar and the grey navigation elements have the widths you want. If your browser window is wide enough, you will also see that the whole page is constrained in width and is centred horizontally.
58. Try changing the window size and see how the layout adapts.

Note: if you are using Microsoft Internet Explorer version 6 or older, you won't see the effects of any width constraints. That's because those versions of IE don't support minimum and maximum widths (or heights). We will look at a workaround for that at the end of the example. In fact, you will get odd results throughout this example, even with IE7, because Internet Explorer has many strange rendering bugs. I will focus on the standards-compliant way to do things in the example, and turn to workarounds at the end.

If you look closely at the code you'll see that the widths were set to 13em and 11em instead of 14em and 12em. That's because you need some horizontal padding; you don't want the content of those columns to lie flush with the edges, because it doesn't look very nice. Padding adds to the width, so 13em + 0.5em + 0.5em adds up to the 14em you want.

Making columns

Okay, you have your basic building blocks, but they just appear one after the other. You want three columns, so you need to start floating them.

1. Add the following rules to your CSS file:

```
2.     #main {
3.         float: left;
4.     }
5.
6.     #sidebar {
7.         float: left;
8.         width: 13em;
9.         padding: 0 0.5em;
10.        background-color: #ff6;
11.    }
12.
13.    #nav {
14.        float: left;
15.        width: 11em;
16.        padding: 0 0.5em;
17.        background-color: #ddd;
18.    }
19. }
```

That floats them, all right, but they're in the wrong order. Also, the main content column is too narrow. And what happened to our footer?

18. Let's deal with the footer first. The problem is that the three columns are floated, which takes them out of the document flow. The footer is pushed up against the header and the line box containing the text is shortened so that the word "Footer" appears to the right of the floats. You can remedy this by making sure the footer is cleared from all the floated columns. Add the following rule to the CSS file:

```
19.     #footer {
20.         clear: left;
21.         border-top: 1px solid #369;
22.     }
23. }
```

22. Now for the three columns. This will be done step by step, and it's going to look rather ugly for a while, but don't despair—it'll be sorted out by the end.

The key to this whole trick is the wrapper element. We will set a left and right margin on it that corresponds to the widths of your side columns (the navigation and the sidebar). The main content column will occupy the whole width of the wrapper, while the side columns will be shifted into the space vacated by the margins. Does that sound complicated? Don't worry, I'll take you through it in small increments. First, set up the margins for the wrapper, by adding the following rule to the CSS file:

```
#wrapper {
    margin: 0 14em 0 12em;
    padding: 0 1em;
}
```

Remember that the values in the `margin` shorthand property are specified in TROUBLE order: top, right, bottom, left. We are setting the top and bottom margins to 0, the right margin to 14em (for the sidebar) and the left margin to 12em (for the navigation). You've also added 1em of horizontal padding, because you don't want your content to be flush with the side columns; it needs to breathe.

23. The next step is to make the main content column take up the full width of its wrapper parent; the code also sets a garish background colour to it, temporarily, so we're doing:

```
24.     #main {
25.         float: left;
26.         width: 100%;
27.     }
```

```
27.      background-color: lime;
    }
```

28. Save and reload—you'll see a bright lime green content column, with the sidebar and navigation below it. You'll also notice that there is a lot of white space on both sides. The trick is to get our side columns to slip into that white space.

Next I'll move you on to the sidebar—it's floated and it has the right width, but since the #main column is 100% wide, it pushes the sidebar down. How do you get it to go up and stay next to #main, although #main occupies the whole width? Let's do it in two small steps: first, you'll move it up; then you'll shift it out into the margin.

29. Here you'll use a nifty trick to get the floated sidebar, which has been pushed down, to move back up again—make the following addition to the #sidebar rule:

```
30.      #sidebar {
31.          float: left;
32.          width: 13em;
33.          padding: 0 0.5em;
34.          background-color: #ff6;
35.          margin-left: -14em;
    }
```

36. Save and reload, and you'll see that the sidebar is now on the same vertical level as the content column. By setting a negative left margin equal to the width of the sidebar, we move the element back into the wrapper and it isn't pushed down. The problem is that it overlaps the content.

37. You need to shift it out into the margin without making it drop down again, and this is where relative positioning—finally—comes in. It does precisely what we want: it shifts the generated box without moving the element itself. Add the highlighted properties below into the rule for #sidebar:

```
38.      #sidebar {
39.          float: left;
40.          width: 13em;
41.          padding: 0 0.5em;
42.          background-color: #ff6;
43.          margin-left: -14em;
44.          position: relative;
45.          left: 15em;
    }
```

Note that you had to shift it 15em, not 14em—that's because there's 1em of right padding on the wrapper that you need to get past. The sidebar is now where it belongs: out in the margin, next to the content column, lining up nicely with the right-hand edges of the header and the footer.

46. Now you need to do the same with the navigation this is done in a similar way, but it has a twist of its own. Moving and shifting the sidebar was easy, because the movements were essentially the same as the column's width: 14em negative margin and a 14em+1em shift to the right. But the navigation column needs to be moved all the way across the content column and then be shifted even further out into the margin.

Our friend here is percentages. A percentage value on the margins of the navigation column will be relative to the width of its parent, the wrapper. You want to move the column all the way across the wrapper—add the property highlighted below to the rule for #nav:

```
#nav {
```

```
float: left;
width: 11em;
padding: 0 0.5em;
background-color: #ddd;
margin-left: -100%;
}
```

47. Hey presto! Save and reload again, and you should see the navigation overlapping the left-hand side of the content column. All you need to do now is to shift it out into the margin. Add the following highlighted properties to the rule for #nav:

```
48. #nav {
49.     float: left;
50.     width: 11em;
51.     padding: 0 0.5em;
52.     background-color: #ddd;
53.     margin-left: -100%;
54.     position: relative;
55.     right: 13em;
}
```

Again, the width of the navigation is 12em, but you have 1em of wrapper padding to get past so you need to shift the box 13em. You're shifting it to the left, in other words *from* the right edge, which is why the `right` property is being used.

56. Remove the lime green background from the content column, and you're all set to go.

Working around quirks in Internet Explorer

There are two properties of this layout that cause it to fail in Internet Explorer 6 for Windows. One is that IE6 doesn't support the `min-width` and `max-width` properties, the other is that IE is notoriously bad at percentages.

You can use Microsoft's proprietary `expression()` notation to emulate the width constraints. It takes a JScript expression as its argument and returns the return value of that expression. This can cause performance problems if the expression requires a lot of computing, since it is evaluated every time the browser needs to get the width of `body`. It also requires JScript to be enabled, but you can add graceful degradation, so that if say, JScript is not available, the design will fallback to something that is still usable. In this example, you'll make the layout fully elastic instead of the constrained fluid design created above if JScript is disabled.

The recommended way of serving bug-fix style rules to Internet Explorer is to make use of "conditional comments". That's a Microsoft-only feature that embeds conditional logic into HTML comments (there is a [dedicated conditional comments article on dev.opera.com](#)).

1. Add the following lines to your HTML code, just before the `</head>` tag:

```
2. <!-- [if lte IE 6]>
3. <link rel="stylesheet" type="text/css" href="layout-ie6.css">

<![endif]-->
```

4. Next, create a new file named `layout-ie6.css` with the following content:

```
5. body {
6.     width: 50em;
7.     width: expression(w=document.documentElement.offsetWidth,
em=document.getElementById("nav").offsetWidth/12,
(w<40*em?"40em"🤪w>56*em?"56em":"auto")));
```

```
8.      }
9.
10.     #wrapper {
11.         height: 1em;
12.     }
13.
14.     #nav {
15.         margin-left: -22em;
16.         margin-left: expression((-
17.             (document.getElementById("wrapper").clientWidth))+"px");
18.         left: 13em;
19.     }
20. }
```

This sorts out the two problems in IE6. You're using JScript expressions to emulate the `min-width` and `max-width` properties that IE6 doesn't support, with an elastic fallback value of 50em. Then you use another JScript expression to set a left margin in pixels instead of percents, again with an elastic fallback. The height for `#wrapper` is just to trigger the Microsoft-specific `hasLayout`, which it needs to have for the relative positioning of the navigation element to work properly. Microsoft has documented `hasLayout` on MSDN, but it's not all that easy to understand.

What about IE7 then? It does support `min-width` and `max-width`, but it still positions the navigation element wrong—it's the same `hasLayout` bug as in IE6 rearing its ugly head again. You need to trigger `hasLayout` on the `#wrapper` element. Fortunately, you can do this in a way that doesn't compromise standards-compliant browsers, so you don't need to create a separate IE7 style sheet; you could just add the following rule to manipulate the wrapper:

```
#wrapper {
    margin: 0 14em 0 12em;
    padding: 0 1em;
    min-height: 1em;
}
```

Setting a minimum height triggers `hasLayout` and it causes no problem in other browsers, so it can go in your main style sheet.

These workarounds aren't perfect; the layout will still do odd things in IE6 and IE7 if the browser window is resized to certain sizes, although if the page is then reloaded, the layout looks okay again.

Other uses for relative positioning

The most common use for relative positioning doesn't involve shifting the generated box at all. This may sound strange: why should you want to use relative positioning without shifting the box? The reason will be revealed in the next article, because it involves absolute positioning. Stay tuned!

Setting `position: relative` (without shifting the box) also helps with some of the strange rendering bugs in Internet Explorer. It sets the infamous `hasLayout` internal property, which has a profound impact on how Internet Explorer renders elements.

Summary

Static positioning is the default state of affairs. Block boxes are laid out vertically in source order, while inline boxes are laid out horizontally in line boxes within those block boxes.

Relative positioning allows you to shift the generated box in one or two dimensions. The element still occupies space as if it were static, but the generated box can be shifted to another position. Relative positioning is mainly useful in combination with floats to create layouts where the presentational order differs from the source order.

Exercise questions

- What happens when two adjacent margins in a static formatting context collapse and one of the margins—or both—is negative?
- Add a vertical border between each of the side columns and the main content column. Remember that all three columns are floated, so the wrapper element's height has collapsed to zero.
- How can you make all the columns have the same height (or at least appear to), so that the background colours extend down to the footer, no matter which column is the longest? (Hint: search for “faux columns” in your favourite search engine.)

About the author



Tommy Olsson is a pragmatic evangelist for web standards and accessibility, who lives in the outback of central Sweden. He wrote his first HTML document in 1993 and is currently the technical webmaster for a Swedish government agency.

He has written one book so far—The Ultimate CSS Reference (with Paul O'Brien)—and has a sadly neglected blog called [The Autistic Cuckoo](#).

37: CSS absolute and fixed positioning

BY TOMMY OLSSON · 26 SEP, 2008

Published in: [Z-INDEX](#), [STACKING](#), [CONTAINER](#), [BLOCK](#), [CONTAINING](#)

This is Article 37 of the Opera Web Standards Curriculum

[Previous article—CSS static and relative positioning](#)

[Table of contents](#)

Introduction

Now it's time to turn your attention to the second pair of `position` property values—`absolute` and `fixed`. The first pair of values—`static` and `relative`—are closely related, and we looked into those in great detail in [the last article](#).

Absolutely positioned elements are removed entirely from the document flow. That means they have no effect at all on their parent element or on the elements that occur after them in the source code. An absolutely positioned element will therefore overlap other content unless you take action to prevent it. Sometimes, of course, this overlap is exactly what you desire, but you should be aware of it, to make sure you are getting the layout you want!

Fixed positioning is really just a specialized form of absolute positioning; elements with fixed positioning are fixed relative to the viewport/browser window rather than the containing element; even if the page is scrolled, they stay in exactly the same position inside the browser window.

In this article I'll give you some practical examples of using both `absolute` and `fixed` positioning, look at some browser support quirks, and explore the concept of z-index.

The article structure is as follows:

- [Containing blocks](#)
- [Absolute positioning](#)
 - [Specifying the position](#)
 - [Specifying dimensions](#)
 - [The third dimension—z-index](#)
 - [Local stacking contexts](#)
- [Fixed positioning](#)
- [Summary](#)
- [Exercise questions](#)

Before I talk about all this though, I'll cover an essential prerequisite concept—containing blocks.

Containing blocks

An essential concept when it comes to absolute positioning is the *containing block*: the block box that the position and dimensions of the absolutely positioned box are relative to.

For static boxes and relatively positioned boxes the containing block is the nearest block-level ancestor—the parent element in other words. For absolutely positioned elements however it's a little more complicated. In this case the containing block is the nearest *positioned* ancestor. By “positioned” I mean an element whose `position` property is set to `relative`, `absolute` or `fixed`—in other words, anything except normal static elements.

So, by setting `position: relative` for an element you make it the containing block for any absolutely positioned descendant (child elements), whether they appear immediately below the relatively positioned element in the hierarchy, or further down the hierarchy.

If an absolutely positioned element has no positioned ancestor, then the containing block is something called the “initial containing block,” which in practice equates to the `html` element. If you are looking at the web page on screen, this means the browser window; if you are printing the page, it means the page boundary.

Elements with fixed positioning differ from this slightly—they *always* have the initial containing block as their containing block.

So, let’s summarize this in a set of easy steps—to find the containing block for an element with `position:absolute`, this is what you need to do:

1. Look at the parent element of the absolutely positioned element—does that element’s `position` property have one of the values `relative`, `absolute` or `fixed`?
2. If so, you’ve found the containing block.
3. If not, move to the parent’s parent element and repeat from step 1 until you find the containing block or run out of ancestors.
4. If you’ve reached the `html` element without finding a positioned ancestor, then the containing block is the `html` element.

Absolute positioning

Fixed positioning is a special form of absolute positioning, so we’ll study that later, and concentrate on the more generalized case here. Unless otherwise stated, when I use the term “absolutely positioned” from now until the end of the article, I’ll be referring both to elements with `position:fixed` *and* elements with `position:absolute`.

Specifying the position

With relative positioning, you learned that the `top`, `right`, `bottom` and `left` properties could be used to specify the position of the box. You use the same properties to specify the position of an absolutely positioned box, but the way you use them is quite different.

For a relatively positioned element, the four properties specify the relative distance to shift the generated box. Remember that in the case of relative positioning they complement one another, so that `top:1em` and `bottom:-1em` means the same, and it’s not meaningful to specify both `top` and `bottom` (or `left` and `right`) for the same element, because one of the values will be ignored.

These points are not true in the case of absolute positioning. Here, all four properties can be used at the same time, to specify the distance from each edge of the positioned element to the corresponding edge of the containing block. You can also specify the position of one of the corners of the absolutely positioned box—say by using `top` and `left`—and then specify the dimensions of the box using `width` and `height` (or just use no `width` and `height` if you want to let it shrink-wrap to fit its contents).

Microsoft Internet Explorer version 6 and older don’t support the method of specifying all four edges, but they do support the method of specifying one corner plus the dimensions.

```
/* This method works in IE6 */
#foo {
  position: absolute;
  top: 3em;
  left: 0;
  width: 30em;
  height: 20em;
}

/* This method doesn't work in IE6 */
#foo {
  position: absolute;
  top: 3em;
  right: 0;
```

```
    bottom: 3em;
    left: 0;
}
```

The thing to remember here is that the values you set for the `top`, `right`, `bottom` and `left` properties specify the distances from the element's edges to their *corresponding* containing block edges. It's not like in a co-ordinate system where each value is relative to one point of origin. For instance, `right: 2em` means that the right edge of the absolutely positioned box will be 2em from the right edge of the containing block.

It's absolutely crucial to know what your containing block is when you're using absolute positioning. That's why setting `position: relative` on your containing block is so useful, even if you are not actually shifting the position of the box. It allows you to make an element the containing block for its absolutely positioned descendants—it gives you control.

Let's try an example out to see how it works.

1. Copy the code below into your text editor and save the document as `absolute.html`.

```
2.    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
3.    <html>
4.        <head>
5.            <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
6.            <title>Absolute Positioning</title>
7.            <link rel="stylesheet" type="text/css" href="absolute.css">
8.        </head>
9.        <body>
10.            <div id="outer">
11.                <div id="inner"></div>
12.            </div>
13.        </body>
    </html>
```

14. Next, copy the following code into a new file and save it as `absolute.css`.

```
15.    html, body {
16.        margin: 0;
17.        padding: 0;
18.    }
19.
20.    #outer {
21.        margin: 5em;
22.        border: 1px solid #f00;
23.    }
24.
25.    #inner {
26.        width: 6em;
27.        height: 4em;
28.        background-color: #999;
    }
```

29. Save both files and load the HTML document into your browser. You will see a grey rectangle surrounded by a somewhat wider red border. The `#inner` element has a specified width and height and a grey background colour. The `#outer` element, which is the structural parent of `#inner`, has a red border. It also has a 5em margin all around, to shift it away from the edges of the browser window and let us see more clearly what is going on.

Nothing surprising so far, right? The height of the `#outer` element is given by its child element (`#inner`) and the width by the horizontal margins.

30. Now watch what happens if you make `#inner` absolutely positioned! Add the following highlighted declaration to the `#inner` rule:

```
31.     #inner {  
32.         width: 6em;  
33.         height: 4em;  
34.         background-color: #999;  
35.         position: absolute;  
    }
```

36. Save and reload. Instead of a red border around the grey rectangle, there is now what looks like a thicker top border only. And the grey box didn't move at all! Did you expect that?

There are two interesting things happening here. First of all, making `#inner` absolutely positioned removed it entirely from the document flow. That means its parent, `#outer`, now has no children that are in the normal flow, so therefore its height collapses to zero. What looks like a 2px-thick red line is actually a 1px border around an element with zero height—you're seeing the top and bottom borders with nothing inbetween.

The second interesting thing is that the absolutely positioned box didn't move. The default value for the `top`, `right`, `bottom` and `left` properties is `auto`, which means the absolutely positioned box will appear exactly where it would have had if it wasn't positioned. Since it's removed from the flow it will overlap any elements in the normal flow that follow it, though.

This is actually very useful—you can rely on this if you only want to move a generated box in one dimension. For instance, in a CSS-driven drop-down menu, the “drop-down” panes can be absolutely positioned with only the `top` property specified. They will then appear at the expected co-ordinate along the X axis (the same as their parent), automatically.

37. Next, let's set a height for the `#outer` element so that it looks like a rectangle again, and move `#inner` sideways. Add the following highlighted lines to your CSS rules:

```
38.     #outer {  
39.         margin: 5em;  
40.         border: 1px solid #f00;  
41.         height: 4em;  
42.     }  
43.  
44.     #inner {  
45.         width: 6em;  
46.         height: 4em;  
47.         background-color: #999;  
48.         position: absolute;  
49.         left: 1em;  
    }
```

50. Save and reload, and you'll see some changes. The `#outer` element is now a rectangle again, since you set a height for it. The `#inner` element has shifted sideways, but not to where you might have expected it to go. It's not 1em from the left border of its parent, but 1em from the left edge of the window!

The reason is that, as explained above, `#inner` has no positioned ancestor, so its containing block is the initial containing block. If you specify a position other than `auto`, it's relative to the corresponding edge of the containing block. When you set `left: 1em`, the left edge of `#inner` ended up 1em from the left edge of the window.

51. If you want it 1em from the left edge of its parent element instead, you must make the parent the containing block. To do this, you'll now use the trick I mentioned earlier in this article—making the parent block relatively positioned. Add the following highlighted line to the #outer rule:

```
52.     #outer {  
53.         margin: 5em;  
54.         border: 1px solid #f00;  
55.         height: 4em;  
56.         position: relative;  
  
    }
```

57. Save and reload—lo and behold! The grey rectangle is now 1em from the left border of the parent element. Setting `position: relative` on the #outer rule has made it positioned and set it as the containing block for any absolutely positioned descendants it might have. The `left: 1em` you set for #inner now counts from the left edge of #outer, not the left edge of the browser window.

Specifying dimensions

Absolutely positioned elements will shrink-wrap to fit their contents unless you specify their dimensions. You can specify the width by setting the `left` and `right` properties, or by setting the `width` property. You can specify the height by setting the `top` and `bottom` properties, or by setting the `height` property.

Any of these six properties can be specified as a percentage. Percentages are, by their very nature, relative to something else. In this case they are relative to the dimensions of the containing block.

For an absolutely positioned element, percentage values for the `left`, `right` and `width` properties are relative to the width of the containing block. Percentage values for the `top`, `bottom` and `height` properties are relative to the height of the containing block.

Internet Explorer 6 and older, and also Opera 8 and older, got this entirely wrong and used the dimensions of the *parent* block instead. Let's experiment with another example to see how that can make a big difference.

1. Begin by specifying the dimensions of #inner using percentage values—make the following changes to the #inner rule:

```
2.     #inner {  
3.         width: 50%;  
4.         height: 50%;  
5.         background-color: #999;  
6.         position: absolute;  
7.         left: 1em;  
  
    }
```

8. Save and reload, and you'll see that the grey rectangle becomes wider and shorter (at least if you're using a modern browser). The containing block is still #outer, since it has `position: relative`. The #inner element's width is now half that of #outer, and its height is half the height of #outer.
9. Let's make the viewport the containing block again, to see the difference! Make the following change to #outer:

```
10.    #outer {  
11.        margin: 5em;  
12.        border: 1px solid #f00;  
13.        height: 4em;  
14.        position: static;
```

}

15. Save and reload—quite a difference, eh? The grey box is now half as wide and half as tall as the browser window.

As you can see, knowing your containing blocks is absolutely essential!

The third dimension—z-index

It's natural to regard a web page as two-dimensional. Technology hasn't evolved far enough that 3D displays are commonplace, so we have to be content with width and height and fake 3D effects. But CSS rendering actually happens in three dimensions! That doesn't mean you can make an element hover in front of the monitor—yet—but you can do some other useful things with positioned elements.

The two main axes in a web page are the horizontal X axis and the vertical Y axis. The origin of this coordinate system is in the upper left-hand corner of the viewport, ie where both the X and Y values are 0.

But there is also a Z axis, which we can imagine as running perpendicular to the monitor's surface (or to the paper, when printing). Higher Z values indicate a position “in front of” lower Z values. Z values can also be negative, which indicate a position “behind” some point of reference (I'll explain this point of reference in a minute).

Before we continue, I should warn you that this is one of the most complicated topics within CSS, so don't get disheartened if you don't understand it on your first read.

Positioned elements (including relatively positioned elements) are rendered within something known as a *stacking context*. Elements within a stacking context have the same point of reference along the Z axis. I'll explain more about this below. You can change the Z position (also called the *stack level*) of a positioned element using the *z-index* property. The value can be an integer number (which may be negative) or one of the keywords *auto* or *inherit*. The default value is *auto*, which means the element has the same stack level as its parent.

You should note that you can only specify an *index* position along the Z axis. You can't make one element appear 19 pixels behind or 5 centimetres in front of another. Think of it like a deck of cards: you can stack the cards and decide that the ace of spades should be on top of the three of diamonds—each card has its stack level, or Z index.

If you specify the *z-index* as a positive integer, you assign it a stack level “in front of” other elements within the same stacking context that have a lower stack level. A *z-index* of 0 (zero) means the same as *auto*, but there is a difference to which I will come back in a minute. A negative integer value for *z-index* assigns a stack level “behind” the parent's stack level.

When two elements in the same stacking context have the same stack level, the one that occurs later in the source code will appear on top of its preceding siblings.

There can in fact be no less than seven layers in one stacking context, and any number of elements in those layers, but don't worry—you are unlikely to have to deal with seven layers in a stacking context. The order in which the elements (all elements, not only the positioned ones) within one stacking context are rendered, from back to front is as follows:

1. The background and borders of the elements that form the stacking context
2. Positioned descendants with negative stack levels
3. Block-level descendants in the normal flow
4. Floated descendants
5. Inline-level descendants in the normal flow
6. Positioned descendants with the stack level set as *auto* or 0 (zero)
7. Positioned descendants with positive stack levels

The highlighted entries are the elements whose stack level we can change using the *z-index* property.

This whole thing can be rather difficult to imagine, so let's do some practical experiments to illustrate Z-index.

1. Begin by adding the following highlighted line to your little sample document:

```
2.     <body>
3.         <div id="outer">
4.             <div id="inner"></div>
5.             <div id="second"></div>
6.         </div>

</body>
```

7. Next, I'll get you to restore your CSS so that #outer is the containing block and set non-percentage dimensions of #inner. Let's make #outer a little taller, too, to give you more room to experiment. Make the following highlighted changes to the two rules:

```
8.     #outer {
9.         margin: 5em;
10.        border: 1px solid #f00;
11.        height: 8em;
12.        position: relative;
13.    }
14.
15.    #inner {
16.        width: 5em;
17.        height: 5em;
18.        background-color: #999;
19.        position: absolute;
20.        left: 1em;
    }
```

21. Add a rule for the #second element, too:

```
22.    #second {
23.        width: 5em;
24.        height: 5em;
25.        background-color: #00f;
26.        position: absolute;
27.        top: 1em;
28.        left: 2em;
    }
```

29. Save and reload, and you'll see a bright blue box overlapping a grey one. Both boxes have the same stack level (auto, the initial value, which means stack level 0) but the blue box is rendered in front of the grey box, because it appears later in the source code. You can make the grey box appear in front by giving it a positive stack level. You only have to set it larger than 0—there's no need to go overboard and use a value like 10000. Add the following highlighted line to the #inner rule:

```
30.    #inner {
31.        width: 5em;
32.        height: 5em;
33.        background-color: #999;
34.        position: absolute;
35.        left: 1em;
36.        z-index: 1;
    }
```


37. Save and reload, and you will now see the grey box appear in front of the blue box.

Local stacking contexts

The rest of this section discusses local stacking contexts. This probably isn't something you will encounter in your normal design work unless you attempt to do some really advanced things with absolute positioning, but I thought I'd include it for completeness. You can elect to skip this if you wish.

Every element whose `z-index` is specified as an integer establishes a new, "local", stacking context in which the element itself has stack level 0. This is the difference I mentioned before between `z-index: auto` and `z-index: 0`. The former doesn't establish a new stacking context, but the latter does.

When an element establishes a local stacking context, the stack levels of its positioned descendants apply within this local context only. These descendants can be re-stacked with respect to one another, and with respect to their parent, but not with respect to the parent's siblings. It's like the parent forms a "cage" around its descendants, so that they cannot escape from it. The descendants may be moved up and down within this cage, but they can't get out of the cage. The parent and its descendants will form an indivisible unit within the stacking context that surrounds the parent.

Imagine you're sorting out your paperwork before delivering it to the accountant who does your taxes. You have expense reports, receipts, order confirmations and whatnot, and you stack one paper on top of another—to make life easier for you accountant, you insert types of papers that belong together in different envelopes.

A local stacking context is analogous to such an envelope. It keeps related elements together and prevents other elements from being inserted between them. You can sort the contents within each envelope as you like, but that sort order only applies within that envelope and has no bearing on the stack of papers as a whole. Your stack now contains a mix of loose papers (elements with stack level `auto`), and envelopes (elements with an integer stack level). Envelopes with positive stack levels lie on top of the loose papers, while envelopes with negative stack levels appear at the bottom of the pile.

Each time you assign an integer value to the `z-index` property for an element, you create an "envelope" that contains that element and its descendants.

Let's look at how those local stacking contexts work. It may look confusing, but it's really not much different from what you've already seen. If you follow the examples, you should be able to get a feel for how things work.

1. Begin by adding some content to your two inner elements—add the highlighted lines to your HTML document:

```
2.     <div id="inner">
3.         <span></span>
4.     </div>
5.     <div id="second">
6.         <span></span>
```

```
</div>
```

7. Add a CSS rule that will apply to both those `span` elements:

```
8.     span {
9.         position: absolute;
10.        top: 2em;
11.        left: 2em;
12.        width: 3em;
13.        height: 3em;

    }
```

This makes the `span` elements absolutely positioned and sets their positions and dimensions. Wait a second though—`span` elements are inline—how can you specify dimensions for inline elements? The answer is that absolutely positioned elements, like floated elements, automatically generate block boxes.

The positions you specify will apply relative to each `span`'s containing block. Since both `span` elements have an absolutely positioned `div` as a parent, those parents take on the role of containing blocks.

14. Let's now add some colour to the `span` elements so you can see where they appear—add the following rules to your style sheet:

```
15.    #inner span {
16.        background-color: #ff0;
17.    }
18.
19.    #second span {
20.        background-color: #0ff;
21.    }
```

21. Save and reload, and you should see a yellow square in the bottom right-hand corner of the larger grey square, and a cyan-coloured square in the bottom right-hand corner of the larger blue square. The grey and yellow squares appear in front of the blue and cyan squares, since the grey square has `z-index:1`.

22. What if you want the cyan square in front of all the other squares? All you need to do is to give it a higher stack level than the grey square. Actually, it's enough to give it the *same* stack level as the grey square, since the cyan square appears later in the markup. Let's try that—make the following change to your CSS:

```
23.    #second span {
24.        background-color: #0ff;
25.        z-index: 1;
26.    }
```

26. Save and reload. If your browser supports the CSS recommendation properly, the cyan square should now be at the front.

The grey square has `z-index:1`, which means it establishes a local stacking context. In other words, you've created one of those "envelopes" and put the grey square and its yellow child square inside.

Confused yet? The next experiment should make things clearer.

1. Set a high stack level for the yellow square to bring it to the front—make the following change to your CSS:

```
2.    #inner span {
3.        background-color: #ff0;
4.        z-index: 4;
5.    }
```

5. If you save and reload you'll see...no change at all! The stack level we specified for the yellow square applies within the local stacking context established by the grey square—the yellow square is inside an envelope together with its grey parent. You could move the cyan square to the front because its parent (the blue square) doesn't establish a local stacking context—it has an implied `z-index:auto`. The blue square is a loose paper in the stack. The yellow and cyan squares are actually

in little envelopes all by themselves (they have an integer stack level and establish local stacking contexts of their own).

6. If you make the blue square establish a local stacking context, you won't be able to move the cyan square to the front unless you also bring its parent (the blue square) to the front. Let's try it—make the following changes to your CSS:

```
7.     #inner {  
8.  
9.         ...  
10.  
11.         z-index: 2;  
12.     }  
13.  
14.     #second {  
15.  
16.         ...  
17.  
18.         z-index: 1;  
19.     }  
20.  
21.     #second span {  
22.  
23.         ...  
24.  
25.         z-index: 3;  
    }
```

26. Save and reload. Now both the grey square and the blue square establish local stacking contexts, giving us two envelopes. At the bottom of the stack is an envelope with stack level 1, containing two inner envelopes (the blue square and the cyan square). At the top of the stack is an envelope with stack level 2, containing two inner envelopes (the grey square and the yellow square). In the first envelope, the blue square has local stack level 0 so therefore appears behind the cyan square, which has local stack level 3. In the second envelope, the grey square has local stack level 0 so therefore appears behind the yellow square with local stack level 4.

Figure 1 shows the four boxes and the two local stacking contexts from the side, along the Z axis.

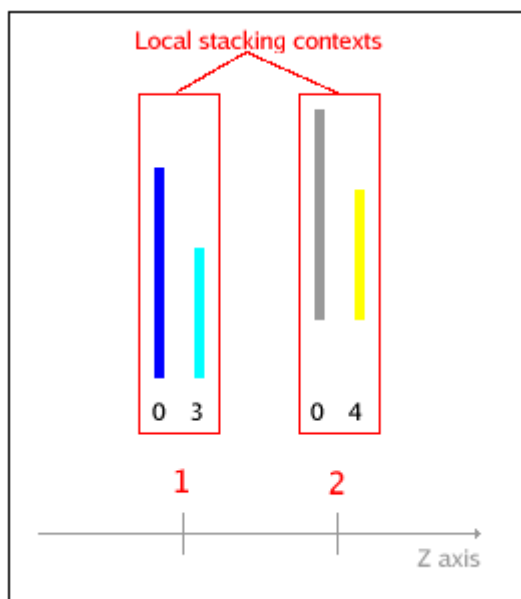


Figure 1: Illustration of different stacking contexts. The elements appearing inside "2" will always appear in front of all of the elements inside "1". Then within each stacking context, elements with a higher z-index number appear in front of elements with a small z-index number. If two elements have the same z-index number, the one appearing later in the markup will appear in front.

This part was probably quite confusing, especially if you're new to CSS. The point is that you need to know your stacking contexts if you're trying to change the stack levels of different elements. If an element belongs to a local stacking context you can only change its position along the Z axis within that local context. An element in one local stacking context cannot slide in between two elements in another local stacking context.

The good news is that you'll most likely never encounter these problems. Changes in `z-index` are not very common in good layouts, and if they occur at all it is usually within one stacking context.

Fixed positioning

An element with `position:fixed` is fixed with respect to the viewport. It stays where it is, even if the document is scrolled. For `media="print"` a fixed element will be repeated on each printed page.

Note that Internet Explorer versions 6 and older do not support fixed positioning at all. If you use one of those browsers you will not be able to see the results of the examples in this section.

Whereas the position and dimensions of an element with `position:absolute` are relative to its containing block, the position and dimensions of an element with `position:fixed` are always relative to the initial containing block. This is normally the viewport: the browser window or the paper's page box.

To demonstrate this, in the example below you will make one of your elements fixed. You will make the other one very tall in order to cause a scrollbar, to make it easier to see the effect it has.

1. Make the following changes to your CSS code:

```
2.     #inner {
3.         width: 5em;
4.         height: 5em;
5.         background-color: #999;
6.         position: fixed;
7.         top: 1em;
8.         left: 1em;
9.     }
10.
11.     #second
12.         width: 5em;
13.         height: 150em;
14.         background-color: #00f;
15.         position: absolute;
16.         top: 1em;
17.         left: 2em;
18. }
```

18. Save and reload. If you don't get a vertical scrollbar, increase the `height` value for `#second`. (What kind of giant monitor do you have, anyway?)

The tall blue element extends beyond the bottom of the window. Scroll the page downward, and keep an eye on the grey square in the top left-hand corner. `#inner` is now fixed in position 1em from the top of the window and 1em from the left side, therefore as you scroll, the grey box stays in the same place on the screen.

Summary

Absolutely positioned elements are removed entirely from the document flow. They will overlap other content unless you make room for them. If all child elements of a container are absolutely positioned, the parent's height will collapse to zero.

Absolutely positioned elements are positioned with respect to a containing block, which is the nearest positioned ancestor. If there is no positioned ancestor, the viewport will be the containing block.

Elements with fixed positioning are fixed with respect to the viewport—the viewport is always their containing block. They always appear at the same place inside the browser window when viewed on screen; when printed, they will appear on each page.

The positions of each edge of an absolutely positioned element can be specified with the `top`, `right`, `bottom` and `left` properties. The value of each property specifies the distance of that edge to the corresponding edge of the element's containing block.

All positioned elements are rendered at a certain stack level within a stacking context. You can change the stack level of a positioned element using the `z-index` property. When `z-index` is specified as an integer value, the element establishes a local stacking context for its descendants.

Exercise questions

- Undo the changes from the fixed positioning example and then change the stacking order between the four absolutely positioned squares so that the grey square is at the back, followed by the blue, yellow and cyan squares in that order. (Tip: remove all `z-index` declarations and start over.)
- Move the yellow square up and to the right by setting `top:-1em` and `left:8em`. Then make it appear *behind* the `#outer` element, so that the red border appears across the yellow square.
- Replicate the three-column layout we created in the [static and relative positioning article](#) using absolute positioning instead. Since it will be impossible to have a full-width footer, you can remove the `#footer` element, but you are not allowed to change anything else in the markup (other than the link to the style sheet).
- Modify the layout from the previous exercise to make the navigation use fixed positioning. You'll have to remove the automatic horizontal margins on the `body` element to make this possible. Add enough content to the main column and/or the sidebar to make a scrollbar appear, so that you can verify that it works.

About the author



Tommy Olsson is a pragmatic evangelist for web standards and accessibility, who lives in the outback of central Sweden. He wrote his first HTML document in 1993 and is currently the technical webmaster for a Swedish government agency.

He has written one book so far—The Ultimate CSS Reference (with Paul O'Brien)—and has a sadly neglected blog called [The Autistic Cuckoo](#).

Supplementary: Getting your content online

BY [CRAIGGRANNELL](#) · 8 JUL, 2008

Introduction

This article aims to provide a quickfire guide for getting your website online. You'll find out about how to get yourself a domain name and some hosting. You'll also find out about the software required to upload websites, along with best practice for structuring a site and ensuring you always have safe copies of your work. This article is split into four sections:

- [What's in a name?](#): How to best select and buy a domain name;
- [The host with the most](#): Covers web-hosting accounts—what to look for and things to be aware of;
- [Getting it on\(line\)](#): Working with software to upload your website(s) on to the Web;
- [Work in progress](#): Working with local and remote files, and best practice for site structure.

What's in a name?

A domain name is an important component when creating a website: it's your text-based link to the world—the thing people type into a browser's address bar to access your site (such as google.com, or apple.com). The best domain names are memorable and straightforward, and therefore superior to the URL you may be given with some free web space, which typically includes your ISP's domain and your broadband or dial-up username (such as nameofisp.com/~username).

When choosing a domain, avoid complexity. Imagine yourself reading it out over the phone—if you have to say things like “hyphen” or “the numeral two”, or if your spelling of a word is awkward or non-standard, think of a different name. As millions of domains are already in use, it pays to be armed with at least a half-dozen alternatives. For example, try being more specific with your name: there's no chance that you'd be able to buy gardening.com, but if you added “in” and your location after “gardening” (such as “gardeninginsomerset.com”), you might.

To find out if a domain is already in use, do a search. The majority of domain resellers have a search form, which enables you to search domains with various suffixes (such as .com, .net, .org, and so on “more on those later”), but a decent, impartial and non-commercial resource is at www.internic.net/whois.html.

If you're on a commercial site doing searches, you might be presented with alternatives should your first choice already be taken. In such scenarios, only settle on alternatives if they're exactly what you're looking for. Don't be tempted by strange word combinations or unusual suffixes. People remember “dot com” or their local equivalent (such as “.co.uk” in the United Kingdom). This is less often the case with awkward combinations like “.uk.com” or newer suffixes such as “.info”. Also, if someone else has a domain with a more popular suffix, you run the risk of losing traffic to them.

Note that even if you're creating a personal site or blog (such as my own Revert to Saved—see Figure 1), it pays to grab a memorable domain. For example, friends and family will find it easier to find your site if it's based around your name, and domains also make associated email accounts more useful, enabling you to use name@yourdomain.com, rather than a borderline random collection of characters prior to your ISP's domain. Also, domains have the advantage of being a constant—should you move ISP and “lose” your free web space, you'll have to start from scratch. However, with a domain name, the address is always the same, meaning that even a move between entirely different web hosts typically only leads to a couple of days of down-time for your site.

When it comes to buying a domain, you have the option of buying it on its own from one of myriad resellers, or buying it alongside a web-hosting account, all from the same organisation. If you're a beginner, I strongly recommend buying your domain and hosting at once. It means you only have a single company to deal with for support issues, and the company's system will likely know what's going on, enabling you to “attach” a newly acquired domain to a hosting account you've purchased via an online administration panel. If you decide against this, you can buy a domain name from one company and “point” it at hosting purchased elsewhere. To do this, you'll need to update your domain's nameservers and IP

address (things that enable the domain to “know” which site it should be pointed at) in line with the requirements of your web host.

Note that pitfalls can occur during the domain buying process. Some sellers inflate prices to make more profit (which is absurd in the current market), and some at the cheaper end of the spectrum charge when you want to move your domain at a later date. Therefore, prior to buying a domain, always check to see whether you can freely move it. Also, don’t be bullied into buying extra domains unless you really need them—if you have your chosen name and a .com, you won’t need a .biz or a .info equivalent as well, so save your money. Finally, if you’re offered private registration during the buying process, it’s worth consideration. By default, your details (name, address, telephone number) will be available when people investigate ownership of your domain; however, for a few bucks, most domain name resellers enable you to “hide” your details, only showing generic details for the registration organisation.

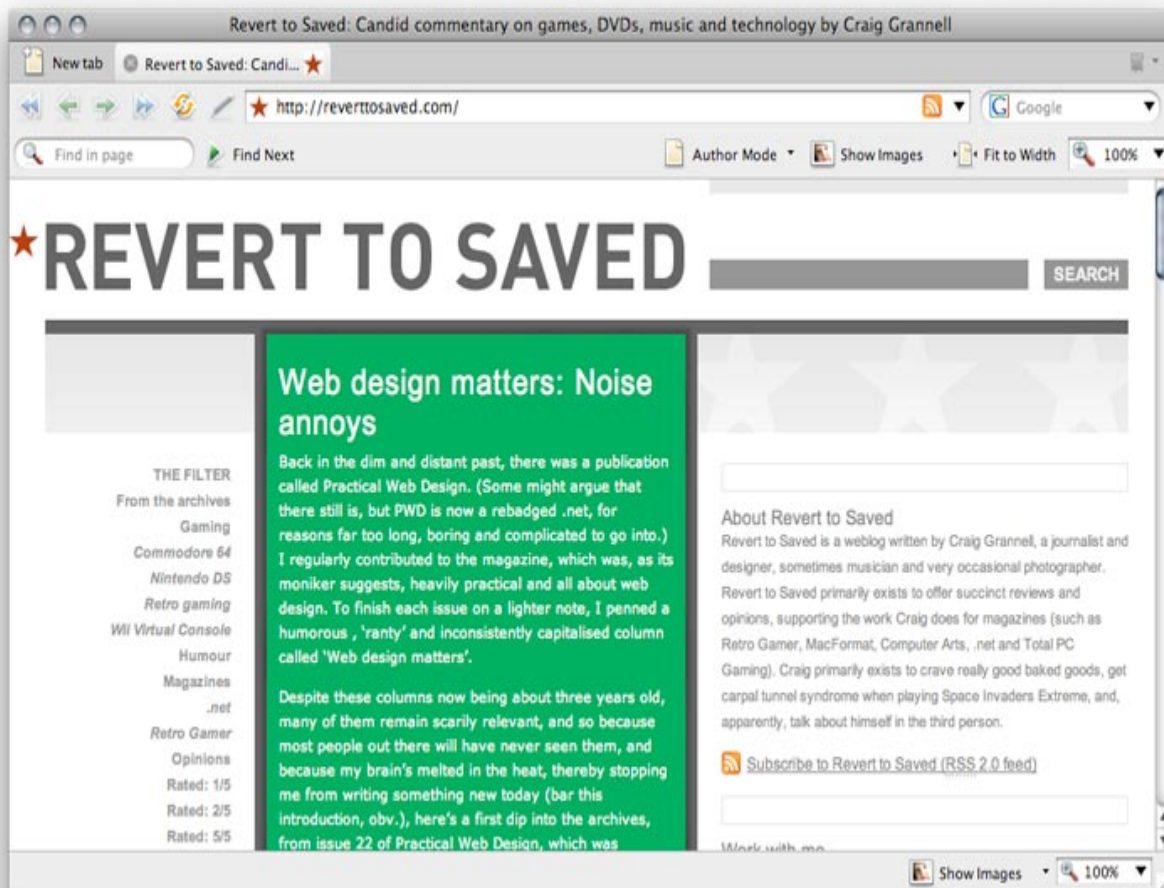


Figure 1: The domain for my blog, [Revert to Saved](http://reverttosaved.com), was carefully chosen. It’s memorable, a string that’s unbroken by hyphens, and has a .com suffix, which is one of the most common.

The host with the most

Once you have a domain name in mind (or already purchased), you need somewhere to upload your site. Chances are you already have free web space with your broadband or dial-up account, but it’s almost certainly restricted in some way, perhaps in terms of bandwidth (the amount of data users can download over a set time period), storage allocation (the number of megabytes or gigabytes you have for storing content), or by way of other technologies (such as support for various types of scripts or for things like

databases). Free hosting tends to offer the bare minimum, meaning you could be scuppered should you later want to expand your site's scope or feature set.

Luckily, improvements in technology have proved to be a good thing for hosting, and competition has pushed down pricing. Cheap hard drives have resulted in paid-for web hosting typically offering masses of space, even on cheap plans. Also, expectations for included features are changing, and so many paid-for hosts provide PHP and MySQL support as a matter of course. In all cases, investigate the technology you're going to need for hosting the kind of site you're building before purchasing; if unsure, talk to your potential host's support, or ensure that a straightforward and affordable upgrade path exists should your requirements change.

There are also further questions to ask. You need to know about levels of data transfer, and what happens if you exceed this level (your site may be temporarily "closed", or you could be charged), although this is only a major concern for very high-traffic websites; you might want to investigate whether readymade items are available, such as preinstalled scripts, forums and contact forms; and if your plans amount to more than a solitary website, you need to be aware of any restrictions relating to hosting multiple domains or sites on a single hosting account. Also, if you're relatively new to setting up sites, it's wise to choose a host where you can speak to an actual human being when you need to, rather than end up stuck waiting for an email to arrive from a semi-automated help system.

A pretty good global tip for hosting is to shop around. Various sites, such as web-hosting-review.toptenreviews.com (see Figure 2), offer opinions and advice regarding the current champions of hosting, and you can also try contacting hosts directly to ask questions. If they respond promptly and suitably, that's a good indication that you'll be in safe hands if problems occur later. In any case, take your time and don't necessarily go for the cheapest option—shop around, do your homework, and ensure the host you settle on is a good match for your needs. As mentioned earlier, though, we recommend first-timers buying their hosting and domain from the same company; usually, you can then "attach" your domain to your hosting account via an online administration panel, in a straightforward and non-technical manner.

Figure 2: Various websites offer comparative reviews of hosting services, and it's a good idea to scour these prior to parting with any cash.

Getting it on(line)

Once you've got your domain and hosting sorted, you can start putting content online. Your web host will provide you with some pieces of information that you'll need to keep safe. You'll likely get details for accessing your account with the host itself, enabling you to access online administration features. You'll also get details for accessing your site via FTP, which stands for "File Transfer Protocol". Although the information provided by web hosts varies, you'll likely be given a username, a password, a location to upload files to (often your URL, but this varies by host), and perhaps a path to the folder where your web pages should be stored. (Note that although you should be able to access your space very quickly via FTP, it can sometimes take up to three days before the entire internet is able to "see" your domain, so don't fret if people can't access your site right away after you've uploaded it.)

	Global	Unipages	PowerWeb	Infology	Easy CGI	Immotion Hosting	Blue Host	Apollo Hosting	FastCow	Netfirms
Reviewer Comments	READ REVIEW	READ REVIEW	READ REVIEW	READ REVIEW	READ REVIEW	READ REVIEW	READ REVIEW	READ REVIEW	READ REVIEW	READ REVIEW
Lowest Price	BUY \$4.95	BUY \$6.95	BUY \$7.95	BUY \$19.95	BUY \$7.96	BUY \$3/mo.	BUY \$6.95	BUY \$6.95	BUY \$8.25	BUY \$9.95
Special Offers	FREE 1st Year after 1st update									
Overall Rating	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★
Ratings										
Feature Set	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★
Customer Service	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★
Control Panel	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★	★★★★★
Costs (based on basic plan for one year)										
Monthly Price	\$4.95	\$6.95	\$7.95	\$6.95	\$7.96	\$7.95	\$7.95	\$7.96	\$8.25	\$9.95
Setup Fee	\$19	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Hosting for 1 year	\$59.40	\$95.40	\$95.40	\$83.40	\$95.52	\$95.40	\$95.40	\$95.40	\$99.00	\$119.40
Features										
Disk Space (GB)	1000	1500	300	500	350	300	200	3	300	250
Monthly Data Transfer (GB)	1000	15000	3000	1000	3500	3000	999	100	3000	2000
Server Uptime Guarantee	99.9%	99.9%	99.9%	99.9%	99.9%	99.9%	99.9%	No Guar.	99.9%	99.9%
Sub-Domain Supported	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Domain Name Search/Registration	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Myriad FTP applications exist, such as the free (but decent) [CoffeeCup Free FTP](#) for Windows, and the excellent [Transmit](#) for Mac OS X. Some web-design applications, such as Dreamweaver, also offer built-in clients, although most aren't as fully featured as a standalone equivalent. FTP applications vary massively, but the majority are roughly comparable in terms of workflow. Typically, you'll have some means of storing favourite locations to connect to (one of which will be your own site). For each favourite, you'll need the details your host provided you with, as mentioned earlier.

Once you've connected to your web space, you'll see the empty folder structure of your website, which also varies by host. In some cases, you'll see nothing at all. In other cases, a few default folders might exist for storing things like scripts and visitor statistics. A good rule of thumb is that you should just leave alone any folders that are there by default. Most FTP clients also provide a local view (as in, that of your hard drive)—see the screen grab of Transmit (Figure 3) for an example. To upload a file to your website, you merely have to drag it from the local to the remote location, or click on the local file and choose a relevant “upload” option.

If you're working with scripts, you may also have instructions to change the permissions of certain files. This is typically done by getting a file's info and clicking relevant checkboxes, but also might be referred to via a “chmod” option—“chmod” being the name of a Unix command for amending file and directory modes. Most FTP clients also offer many more features, including the ability to compare and synchronise local and remote folders and to automatically set a local folder when a favourite is accessed. Again, shop around, and bear in mind that most FTP clients are cheap (or free) and have fully working demo versions.

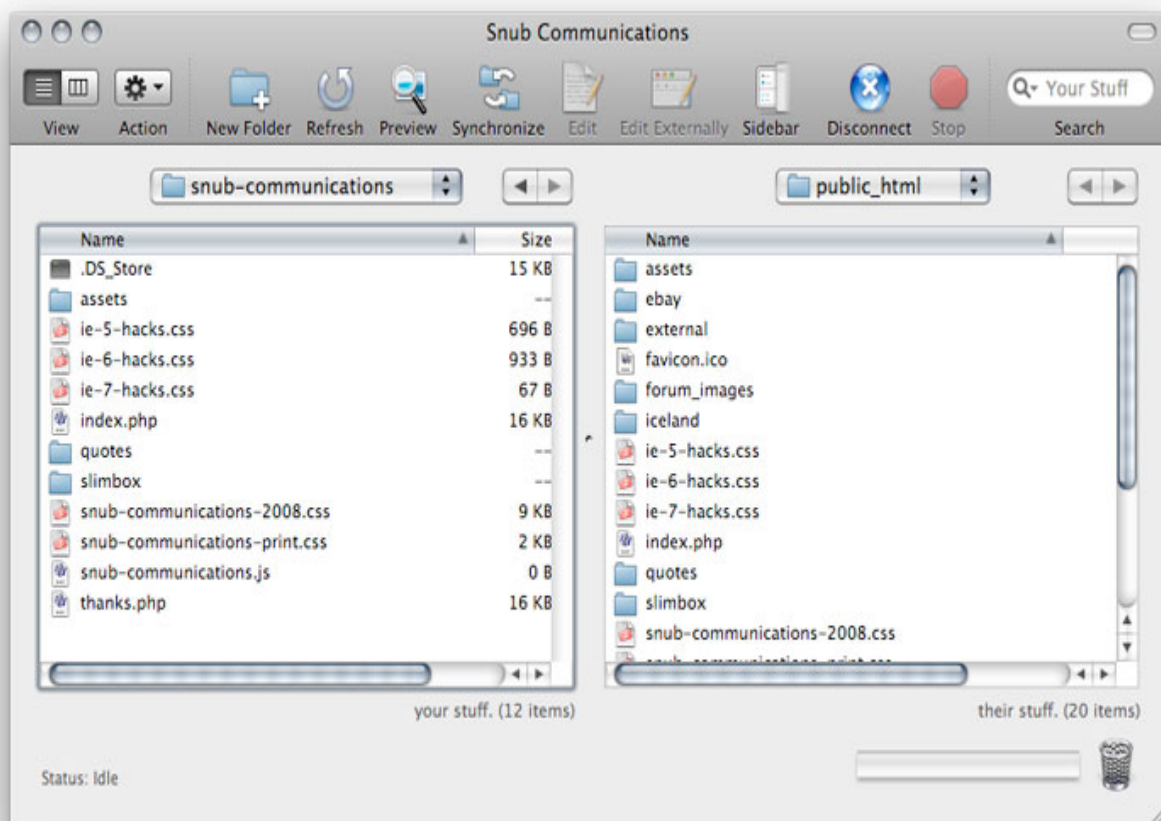


Figure 3: Transmit, available for Mac OS X, is a fairly typical dual-pane FTP client, showing a local view on the left and remote files on the right.

Work in progress

In the previous section, I mentioned how FTP clients often show remote and local files simultaneously. This is a good thing—as any decent web designer will tell you, work solely online at your peril. If you screw up a change when working on a live site, the entire world will see it until a fix is made, and if something happens to the site (hosts do take back-ups, but they aren't always successful, nor as regular as they should be), you lose everything if you're only working online.

Instead, you should work on local copies of your files and only upload them when they're ready. By doing this, you can test changes before they're uploaded, ensuring that they work and that things like text and images are proofed and readable. You can also take back-ups of a site prior to working on major changes, ensuring that if a total disaster occurs, you have a version to fall back on. Only when you're totally happy with your changes should you upload them.

On site structure, it's largely down to the individual how things are organised, but it pays to create a sensible folder structure, enabling you to store things like images, PDFs, MP3s and movies in specific, named folders (see Figure 4), rather than binging everything in the root folder of the site, which can be messy and make things increasingly hard to organise and sort over time. Some web designers also advocate placing stylesheets, JavaScript documents and even groups of web pages into named folders, although this is only really necessary when you have a fairly large number of them. Over time, if the site expands, it may be wise to add sub-folders within folders, to better organise media.

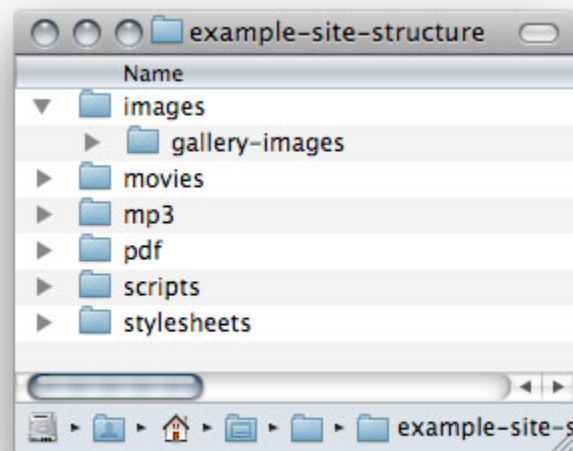


Figure 4: A fairly typical site structure, awaiting content.

From a development standpoint, ensure your local and remote folders are identical in structure, otherwise updating and keeping your “test” and “live” sites consistent will be borderline impossible. (Note also that some hosts request that certain file types be placed in specific folders. The most common example of this is CGI scripts, which often have to be within a *cgi-bin* folder to run. Also, some configuration options—such as for databases—are host-specific. As always, ask your host for advice if unsure.)

Summary

Overall, when it comes to everything mentioned in this overview, caution and research are the two most important considerations. Don't rush headlong into anything and you won't make a costly mistake. Do your research (on domains, hosts, how best to use your web space, and how to upload and maintain your site), and you'll have an easier time of it.

About the author

Originally trained in the fine arts, Craig Grannell became irrevocably immersed in the world of digital media over a decade ago, and he's never looked back. Along with designing sites for a wide range of clients, he's written for various design-oriented publications, penned books on web design, and continues to be creative in the fields of music and photography. Regarding web design, Craig is a stoic cheerleader for both web standards and engaging, simple design.



Find out more about Craig's design and writing via [Snub Communications](#). Craig also regularly writes for his blog [Revert to Saved](#), and occasionally finds the time to release music via [Project Noise](#).

Supplementary: More about the document <head>

BY [CHRISTIAN HEILMANN](#) · 8 JUL, 2008

Introduction

In [Article 13](#) of this course you learned about the essential things that go inside the `head` of an HTML document. In this tutorial I'll expand on that information and talk about some other—lesser used—things that you can add to the `head` section of an HTML document; these are less essential, but still very useful nonetheless. By the end of this tutorial you'll know how to collate several HTML documents into a larger multi-part collection, what a favicon is and how to use it, and what RSS is all about. Before you go any further, [download this article's accompanying zip file](#) so you can follow along with the examples. The contents of this article are as follows:

- [Document relationships—collating several HTML documents into a collection](#)
- [Linking to alternative versions of the document](#)
 - [Translations](#)
 - [Feeds](#)
- [Making bookmarking more fun—using favicons](#)
- [Summary](#)
- [Exercise questions](#)

Document relationships—collating several HTML documents into a collection

One feature of HTML that stems from the origins of the web as a document repository are document relationships. These define how one document relates to another, for example if it is the previous or next document in a logical chain or if it is the index of a whole series of documents.

In a sense, you've already done this in [Article 13](#), when you applied a style sheet to a document to give it a different look and feel with the `link` element:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Breeding Dogs - Tips about Alsations</title>
  <link rel="stylesheet" type="text/css" media="screen" href="styles.css">
  <link rel="stylesheet" type="text/css" media="print" href="printstyles.css">
</head>
<body>
</body>
</html>
```

The relationship of the current document to others is defined in much the same way using the `link` element and the `rel` or `rev` attributes. The `rel` attribute points to a resource that logically comes after the current one, and the `rev` attribute points to a backward resource that you should have come to the current document from, were you to read the collection in the correct order.

There are no mandatory prefixed values for the `rel` and `rev` attributes, but there is a taxonomy supported by browsers and indexing tools that you should consider following under most normal circumstances (you can also use `rel` and `rev` for other purposes, such as Microformats—[check out this article for some uses of the XFN Microformat](#)):

home

The home document of the current collection

index

The index of the current collection

contents

The contents list of the current collection

search

The search page of the current collection

glossary

The glossary of the current collection

help

The help page of the current collection

first

The first document in current collection

previous

The previous document from this one in the logical order of the collection

next

The document following this one in the logical order of the collection

last

The last document of the current collection

up

The document one level up in the hierarchy of the current collection

copyright

The copyright information of the current collection

author

The information page about the author of the current collection

Most browsers don't do anything with this information. Some however will follow the link and load the document in the background so that it shows up a lot faster for the reader. The real browser exception is Opera, which has an extra navigation toolbar you can turn on by selecting View > Toolbars > Navigation bar from the menu. Once turned on you get the link relationships defined in the document as an extra toolbar. Figure 1 shows the W3C HTML standards document in Opera:

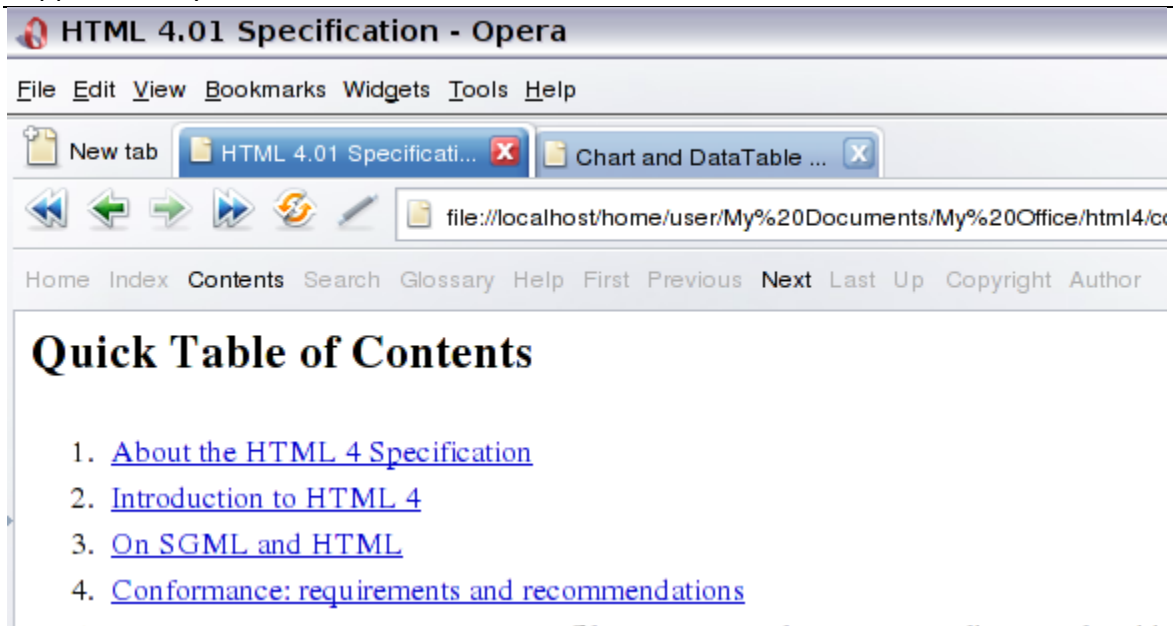


Figure 1: Opera shows the link relationships of the current document in a special navigation toolbar

Even though they are not displayed in a visible sense, it is a good idea to provide a human-readable explanation of what the linked documents are about in a `title` attribute as the file names alone are not necessarily enough.

Now let's move on to have a look at how link relationships can be used to collate several documents into a collection. For example, the start page of an online course spanning several documents could be the following ([start.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 1//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>

  <title>Link relationship example</title>
  <link rel="contents" title="table of contents" href="toc.html">
  <link rel="next" title="next: chapter one" href="chapter1.html">
</head>
<body>
  <h1>Course example</h1>
  <p>This would be the cover page of an article series or course</p>
  <ul>
    <li><a href="chapter1.html" rel="next">Let's start with Chapter One</a></li>
  </ul>
</body>
</html>
```

The first chapter would be the following ([chapter1.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 1//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>

  <title>Chapter One - Link relationship example</title>
  <link rel="contents" title="Table of Contents" href="toc.html">
```



```

<link rel="home" title="Home Page" href="start.html">
<link rel="prev" title="previous: Home Page" href="start.html">
<link rel="next" title="next: Second Chapter" href="chapter2.html">
</head>
<body>
<h1>Chapter One</h1>
<p>This would be the chapter one page of an article series or course</p>
<ul>
<li><a href="start.html" rev="prev">Back to Start</a></li>
<li><a href="toc.html" rel="contents">Table of contents</a></li>
<li><a href="chapter2.html" rel="next">Go on to Chapter Two</a></li>
</ul>
</body>
</html>

```

The second chapter ([chapter2.html](#)):

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 1//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Link relationship example</title>
<link rel="contents" title="Table of Contents" href="toc.html">
<link rel="home" title="Home page" href="start.html">
<link rel="prev" title="previous: first chapter" href="chapter1.html">
</head>
<body>
<h1>Chapter Two</h1>
<p>This would be the second chapter page of an article series or course</p>
<ul>
<li><a href="chapter1.html" rev="prev">Back to chapter 1</a></li>
<li><a href="toc.html" rel="contents">Table of contents</a></li>
</ul>
</body>
</html>

```

And finally the table of contents ([toc.html](#)):

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 1//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Table of contents - Link relationship example</title>
<link rel="home" title="home page" href="start.html">
</head>
<body>
<h1>Table of contents</h1>
<ul>
<li><a href="chapter1.html">Chapter One - about stuff</a></li>
<li><a href="chapter2.html">Chapter One - about other stuff</a></li>
</ul>
<ul>
<li><a href="toc.html" rel="home">Back to home</a></li>
</ul>
</body>
</html>

```

Notice that you can also use `rel` and `rev` attributes on the links in the document to tell browsers and assistive technology that these anchors correspond with the link relationships.

Linking to alternative versions of the document

The option to link to other documents that have a certain relationship to the document in question also includes different language versions of the same document, or different formats. You can do both by providing a link with a `rel` attribute value of `alternate`, indicating an alternative version.

Translations

Translations are a great candidate for document interlinking. It might for example be that one language version of a document is very successful and visitors who don't speak that language would love to have that information available to them. By linking from the original to the alternative language version you'll make it easier for readers of the alternative to understand and promote the content and possibly make the other language version as successful. The following example shows how you can define the other language versions ([languageexample.html](#)); note the syntax—it's pretty intuitive:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 1//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Multiple Languages example</title>
  <link rel="contents" title="table of contents" href="toc.html">
  <link rel="next" title="next: chapter one" href="chapter1.html">
  <link rel="alternate" title="The course in Dutch" type="text/html"
hreflang="nl" href="../nl/start.html">
  <link rel="alternate" title="The course in German" type="text/html"
hreflang="de" href="../de/start.html">
</head>
<body>
  <h1>Course example</h1>
  <p>This would be the cover page of an article series or course</p>
  <ul>
    <li><a href="chapter1.html" rel="next">Let's start with Chapter One</a></li>
  </ul>
  <ul>
    <li>Other languages:
      <ul>
        <li><a href="../de/start.html" lang="de" hreflang="de">Deutsch</a></li>
        <li><a href="../de/start.html" lang="nl"
hreflang="nl">Nederlands</a></li>
      </ul>
    </li>
  </ul>
</body>
</html>
```

There is much more to explore with offering an international version of a web site than this, and we are hoping to provide a dedicated tutorial on this subject later on in the course. You might have noticed the attributes `hreflang` and `lang` that you might not have seen before. The `hreflang` attribute on links and anchors defines the human language of the linked document and the `lang` attribute defines the language of the text inside the element that has this attribute. This is very important for accessibility as text-to-speech software needs to switch the pronunciation voice from language to language.

Different languages have obviously been around since the Internet first existed (and thousands of years before that,) but there is another type of alternative web page that you'll see a lot as you trawl the web—feeds (eg RSS feeds). These are very popular, especially for documents that change constantly, such as news sites. I'll look at these next.

Feeds

A feed is a document containing condensed information detailing the new additions to your site in chronological order. Users can subscribe to it and get to know what has changed on your site recently without having to visit it. They do this by using tools like feed readers, eg [Google Reader](#), [Netvibes](#) or [Bloglines](#). Some modern browsers (such as Opera) and e-mail clients (such as Mac Mail, or Outlook on

Windows) can also process and display feeds. You can recognize that a web site offers a feed by the RSS icon next to the location as shown in Figure 2:

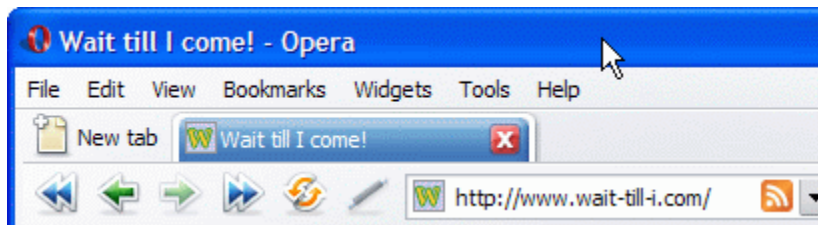


Figure 2: Opera shows an orange RSS icon next to the location of web sites that offer a feed.

Feed pages are either structured using HTML or an XML format like RSS or Atom, and they are hardly ever generated by hand. Most of the time personal publishing systems will do that work for you and all you need to do to offer the world a feed of your site is link to the XML document with the correct meta element in the head of your document. The following is an excerpt from my blog at <http://wait-till-i.com> and points to the RSS feed ([feedexample.html](http://wait-till-i.com/feed/)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 1//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <link rel="alternate" type="application/rss+xml" title="Wait till I come! RSS
Feed" href="http://www.wait-till-i.com/feed/">
  <title>Wait till I come!</title>
</head>
<body>
</body>
</html>
```

Supplying a feed makes sense for content-heavy web sites that change very often (like blogs or photo sites), and by using a feed reading tool and subscribing to feeds you can cut down on a lot of your surfing and research time.

If you don't update your site that often but you have a lot of content and want people to have a visual reminder of your web site, then you might want to consider using a shortcut icon to stand out in people's bookmark lists. This is what I'll cover in the section below.

Making bookmarking more fun—using favicons

One last subject I'll cover here is shortcut icons or favicons. These are small images with a file format of .ico—if you place one on your web server, you can use it to show a small icon next to the entry of your web site in a visitor's bookmark list, as shown in Figure 3:

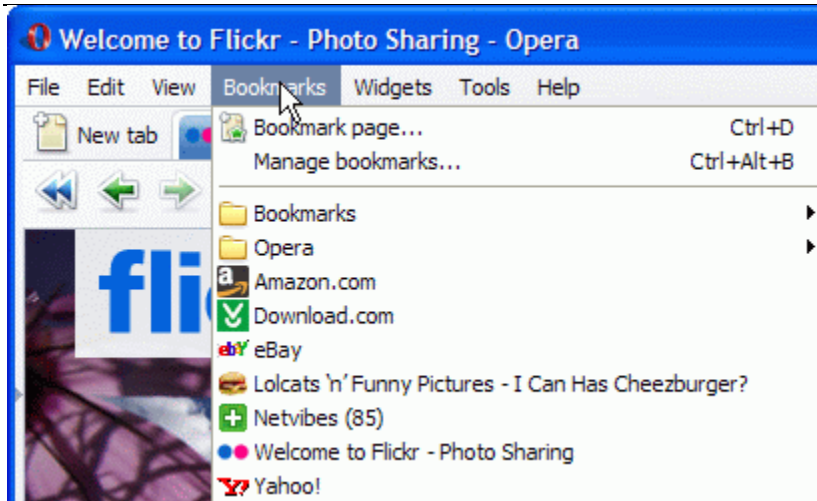


Figure 3: Icons next to a bookmark make it easier to remember the site. You can add one of these by using a shortcut-icon meta element.

The biggest obstacle to adding your shortcut icon is actually creating it in the right format as not many graphics creation packages support the ico format. One option is to use the free online tool [genfavicon](#). Once you have it, adding it to your document is as easy as adding another meta element with a `rel` value of “Shortcut Icon”, as shown in the following example ([favicon-example.html](#)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 1//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Shortcut Icon example</title>
  <link rel="Shortcut Icon" href="favicon.ico" type="image/x-icon">
</head>
<body>
  <h1>Example of a shortcut icon</h1>
</body>
</html>
```

If you open this document in a browser it should show the Opera icon next to the address in the location toolbar. If you bookmark it, the same icon will appear next to the bookmark.

Summary

That’s all for this article, and for the `head` section of an HTML document. There are other things I could cover, but they tend to be fairly advanced subjects, and often not a good idea—what I have covered here, and in [Article 13](#), should give you all you need to get on. In this article I covered:

- Defining document relationships with the `rel` and `rev` attribute in `link` elements
- Linking to alternative versions of the same document like translations or feeds
- Adding a shortcut icon to documents that shows in bookmarks and in browser tabs

Exercise questions

- Why does it make sense to define link relationships when they are not displayed?
- How would you link to a search page?
- What use is offering a feed to your visitors? What `rel` value do you use to link to one?
- What do you need to make sure of when you link to documents in other languages?

- If you open the example documents in a text editor you'll find another `meta` element we haven't discussed here with an attribute of `content-type`, and something called `utf-8`. What is `utf-8`?

About the author



Photo credit: [Bluesmoon](#)

Chris Heilmann has been a web developer for ten years, after dabbling in radio journalism. He works for Yahoo! in the UK as trainer and lead developer, and oversees the code quality on the front end for Europe and Asia.

Chris blogs at [Wait till I come](#) and is available on many a social network as “codepo8”.